MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO

LEANDRO AMATO LORIATO
LEONARDO AMATO LORIATO

PROTÓTIPO DE ANTI-VÍRUS ESTÁTICO COM VERIFICAÇÃO ON-ACCESS EM AMBIENTE WINDOWS UTILIZANDO A ENGINE DO CLAMAV

Rio de Janeiro
2008

INSTITUTO MILITAR DE ENGENHARIA

LEANDRO AMATO LORIATO
LEONARDO AMATO LORIATO

PROTÓTIPO DE ANTI-VÍRUS ESTÁTICO COM VERIFICAÇÃO ON-ACCESS EM AMBIENTE WINDOWS UTILIZANDO A ENGINE DO CLAMAV

Projeto de Final de Curso apresentada ao Curso de Graduação do Instituto Militar de Engenharia, como requisito parcial para obtenção do título de Graduado em Engenharia de Sistemas e Computação.

Orientadores: Cláudio Gomes de Mello – Maj QEM – D.Sc.
Luiz Henrique da Costa Araújo – Maj QEM – D.Sc.

Rio de Janeiro

2008

INSTITUTO MILITAR DE ENGENHARIA
Praça General Tibúrcio, 80 – Praia Vermelha
Rio de Janeiro - RJ CEP: 22290-270

L872 Loriato, Leandro Amato e Loriato, Leonardo Amato

Protótipo de Anti-vírus Estático com Verificação On-Access Utilizando a Engine do ClamAv / Leandro Amato Loriato e Leonardo Amato Loriato - Rio de Janeiro: Instituto Militar de Engenharia, 2008.

146p.: il, graf., tab.

Projeto de Final de Curso - Instituto Militar de Engenharia - Rio de Janeiro, 2008

1. Segurança da Informação. 2. Anti-vírus. 3. Windows Drivers. I. Leandro Amato Loriato e Leonardo Amato Loriato II. Instituto Militar de Engenharia. III. Título.

CDD 005.8

INSTITUTO MILITAR DE ENGENHARIA

LEANDRO AMATO LORIATO

LEONARDO AMATO LORIATO

PROTÓTIPO DE ANTI-VÍRUS ESTÁTICO COM VERIFICAÇÃO ON-ACCESS EM AMBIENTE WINDOWS UTILIZANDO A ENGINE DO CLAMAV

Projeto de Final de Curso apresentado ao Curso de Graduação em Engenharia de Sistemas e Computação no Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Graduado em Engenharia de Sistemas e Computação.

Orientadores: Cláudio Gomes de Mello – Maj QEM – D.Sc.
Luiz Henrique da Costa Araújo – Maj QEM – D.Sc.

Aprovado em 15 de agosto de 2008 pela seguinte Banca Examinadora:

_____
Luiz Henrique da Costa Araújo – Maj QEM – D.Sc. - Presidente

_____
José Antonio de Sousa Fernandes – Maj QEM – M.Sc.

_____
Paulo Renato da Costa Pereira – Cap QEM – D.Sc.

_____
Anderson Fernandes Pereira dos Santos – Cap QEM

_____
Alex de Vasconcelos Garcia – Prof. – D.Sc.

Rio de Janeiro

2008

Dedicamos este trabalho à memória de nossa querida e amada avó, Apparecida Olegario Amato.

AGRADECIMENTOS

Primeiramente a Deus, por nos ter concedido várias Graças nesta vida.

Aos nossos pais, Umbelino e Ana Loriato, e a todos os nossos familiares, por nos ter dado sempre apoio em vários momentos difíceis durante esse projeto.

Ao nosso primeiro orientador, Maj QEM Cláudio Gomes de Mello, por nos ter guiado nossos primeiros passos durante a construção desse PFC, nos instruindo a procurar conhecer o ClamAv e a procurar detalhes a respeitos dos *File System Filter Drivers*.

Ao nosso segundo orientador, Maj QEM Luis Henrique da Costa Araújo, por nos ter "adotado", por nos dar diversas dicas sobre o ClamAv e, principalmente, por nos manter com os pés no chão durante o desenvolvimento deste projeto.

Ao professor Alex de Vasconcelos Garcia, por nos ajudar diversas vezes com seus conhecimentos acerca de compiladores, solucionando diversos problemas que nos deixaram estagnados semanas, sem saber o que fazer.

Aos professores Cap QEM Anderson Fernandes Pereira dos Santos e Cap QEM Paulo Renato da Costa Pereira, por todo o apoio que nos deram em tantos momentos do desenvolvimento deste projeto e mesmo depois, que não temos nem como enumerar aqui.

Ao nosso colega de turma Diego Barros da Fonseca, pelas dicas de como implementar o suporte multi-língua no anti-vírus.

Ao nosso colega de turma Leonardo Bomfim de Souza, pela ajuda com a interface gráfica, nos orientando para manter o *design* do anti-vírus sempre o mais simples possível.

Por fim, ao Instituto Militar de Engenharia, do qual sentimos muito orgulho de termos sido alunos.

"*Si vis pacem, para bellum*"

- Flavius Vegetius Renatus, escritor e estrategista romano

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

RESUMO

Este Projeto de Fim de Curso tem por finalidade a construção de um protótipo de anti-vírus estático livre para fins de uso do Exército Brasileiro. Esse anti-vírus deverá ser para Windows e residente, isto é, que realize a verificação de vírus na medida em que os arquivos forem sendo acessados pelo sistema operacional. Para o desenvolvimento do protótipo, será utilizado como base a *engine* do anti-vírus livre ClamAv.

ABSTRACT

This Course Conclusion Project aims the construction of a prototype of a free static antivirus for use of the Brazilian Army. This antivirus will be for Windows and will be resident, i.e., it should verify virus as the files are being accessed by the operating system. To develop the prototype, we will be using the engine of the ClamAv free antivirus.

1. INTRODUÇÃO

1.1. CONTEXTO E MOTIVAÇÃO

Ao longo das últimas décadas, temos vivenciado um aumento na capacidade de processamento dos computadores. A Tecnologia de Informação hoje permeia muitas áreas do conhecimento humano e desempenha papel crucial na vida das pessoas, auxiliando em pesquisas científicas, em afazeres domésticos e, especialmente, no processo produtivo de empresas.

Simultaneamente com esse aumento de capacidade computacional, temos experimentado também o crescente aparecimento de *malwares* (*malicious software*), *softwares* destinados a se infiltrar em um computador alheio de forma ilícita, com o intuito de causar algum dano ou roubo de informações.

A defesa dessas ameaças tem sido feitas através de ferramentas como anti-vírus e *anti-spywares*. Tais ferramentas, no entanto são em sua grande e quase absoluta maioria pagas e de código-fechado, conservando o conhecimento da tecnologia necessária para construção dessas ferramentas sobre posse de um grupo muito restrito de empresas no mundo.

O Exército Brasileiro tem se preocupado cada vez mais com a questão da Segurança da Informação em seus sistemas computacionais e na sua comunicação. Por esse motivo, foi incluído um grupo finalístico no seu Plano Básico de Ciência e Tecnologia (PBCT) para tratar especificamente de tais assuntos, o chamado Grupo de Segurança da Informação (GSI).

É questão de importância estratégica a aquisição de conhecimentos específicos nessa área e desenvolvimento de suas próprias tecnologias e ferramentas, visto que, em última instância, não se pode garantir a obtenção desses *softwares*, hoje em poder de apenas alguns países. Faz-se necessário, pois, atingir o quanto antes um estágio de auto-suficiência nessa área. Por tais motivos, o Exército Brasileiro incluiu também em seu PBCT objetivos relacionados à essa questão.

Com relação à parte específica de *softwares* anti-vírus, pouco ou nenhum esforço tem sido feito no sentido de garantir tal auto-suficiência. E foi nesse contexto que este projeto se desenvolveu.

1.2. OBJETIVOS DO PROJETO

Visando suprir a demanda pela auto-suficiência em *softwares* anti-vírus, este trabalho visa desenvolver um protótipo de anti-vírus estático livre e residente para uso do Exército Brasileiro. Tal anti-vírus deverá ser para o sistema operacional Windows, ainda muito utilizado em áreas críticas do EB e notável por ser o objetivo primário dos *malwares* desenvolvidos no mundo.

A plataforma Windows também é notável pela obscuridade com relação a detalhes de seu funcionamento interno e implementação, se comparada com sistemas operacionais livres, como o Linux, por exemplo. É, portanto, objetivo deste trabalho gerar conhecimento sensível sobre o funcionamento do Windows, lançando alguma luz sobre esse assunto e permitindo inclusive, a partir de tal conhecimento, que se desenvolvam diversas outras ferramentas de segurança, como sistemas IDS (*Intrusion Detection System*) , *anti-spywares*, *loggers* de atividades do sistema, entre outros.

1.3. ORGANIZAÇÃO DO RELATÓRIO

O relátorio está dividido em 9 capítulos, sendo o primeiro e o último capítulo referentes respectivamente à introdução e à conclusão do trabalho.

O segundo capítulo apresentará os conceitos básicos de anti-vírus, descrevendo as técnicas de detecção de vírus, bem como a evolução dos mesmos, além de discernir entre os tipos de anti-vírus existentes.

O terceiro capítulo apresentará especificamente o anti-vírus chamado ClamAv, disponível para Linux, que serviu de base para o desenvolvimento deste trabalho, descrevendo detalhes sobre a sua arquitetura neste sistema operacional, sobre sua portabilidade para o Windows e encerrando descrevendo as vantagens e desvantagens de sua adoção.

O quarto capítulo apresentará os conceitos necessários para a construção de um *driver* no Windows. Para tal será apresentada a arquitetura geral do Windows, o modelo de *drivers* utilizado atualmente nesse sistema operacional e as *frameworks* disponíveis para construção propriamente dita dos *drivers*.

O quinto capítulo aborda alguns conceitos e tópicos contidos na API do Windows, que são necessários para a interação direta de um programa com o sistema operacional Winodws em alto nível. Dentre esse tópicos, podemos citar a criação de *Threads*, a comunicação entre processos e a manipulação do Registro do Windows, por exemplo.

O sexto capítulo fala sobre uma linguagem de construção de instaladores, denominada *Null Scriptable Install System* (NSIS). Nesse capítulo, são descritos os principais comandos utilizados na construção do instalador do anti-vírus.

O sétimo capítulo é mais prático, abordando detalhes sobre o protótipo do anti-vírus em si. É nele que descrevemos a arquitetura do sistema, dividindo o anti-vírus em componentes, e também seu funcionamento interno.

Por último, temos o oitavo capítulo que se propõe a descrever as principais funcionalidades do protótipo de anti-vírus e um tutorial passo-a-passo de como utilizar essas funcionalidades. É um capítulo para o usuário comum.

Além dos nove capítulos existentes, existe mais um capítulo com as referências bibliográficas usadas nesse trabalho e um outro, o apêndice, com alguns tópicos importantes para o trabalho, mas que não são necessários a compreensão do mesmo.

2. ANTI-VÍRUS

2.1. CONCEITO E FINALIDADE

Um Anti-vírus é um programa capaz de detectar, identificar, neutralizar e muitas vezes eliminar a presença de determinados programas em um computador denominados *malwares* (*malicious software*).

Rigorosamente tais programas não deveriam ser chamados de Anti-vírus, mas sim "Anti-*malwares*", por lidarem com vários tipos de ameaças e não somente com vírus. Na década de 80, quando este termo foi cunhado, havia apenas os vírus como formas de ameaça, mas atualmente diversos outros tipos são encontrados, como *Worms*, *Trojans*, *Spywares*, *Adwares*, etc.

Tecnicamente um vírus é um programa de computador capaz de se auto-replicar, infectando um arquivo, geralmente um executável, anexando seu conjunto de instruções ao mesmo e, através dessas instruções, infectando outros arquivos e causando algum tipo de dano à máquina.

Worms são também *malwares*, assim como os vírus, capazes de auto-replicação e que causam dano a máquina hospedeira, mas que, diferente dos vírus, não usam arquivos executáveis como meio de propagação, mas falhas e vulnerabilidades de redes, que lhe permitem se conectar a internet e transmitir seus conteúdos a outros *hosts*.

Trojan, em analogia com o artifício do "Cavalo de Tróia" da Mitologia Grega, são programas aparentemente inofensíveis como cartões de mensagens de paz, que ocultam sua real funcionalidade, que é a instalação de um programa no computador alvo que abre determinadas vulnerabilidades (como os chamados *backdoors*), para a posterior invasão por um *cracker*[1] ou mesmo outro programa malicioso.

Spywares são programas maliciosos que se instalam em um computador, ocultando sua existência do usuário, geralmente sem o intuito de causar danos à máquina hospedeira (além do sintoma clássico de lentidão causado pelo *overhead* de processamento da CPU), com a *nobre* tarefa de capturar senhas de e-mail, MSN, Orkut e cartões de crédito.

---

[1] Muitas pessoas utilizam erroneamente o termo *hacker* para descrever criminosos que invadem ou atacam o computador alheio, muitas vezes por culpa da mídia. É importante ressaltar que este não é o termo correto, sendo tais pessoas denominadas *crackers*, um subgrupo dos *hackers*.

Esta última palavra é designada para descrever as pessoas que se interessam pela investigação de *bugs* e vulnerabilidades de programas, gostando de "fuçar" nos mesmos, mas não necessariamente gostando de estragar computadores utilizando esses conhecimentos.

A título de curiosidade, a palavra *hacker* originalmente era usada para descrever os carpinteiros que utilizavam ferramentas robustas como machados para fazer seu trabalho (*hack)*, sendo na década de 40 e 50, estendidas aos radioamadores e, mais tarde, na década de 60, estendidas a programadores.

Adwares são programas semelhantes aos *Spywares*, que ocultam sua existência do usuário, não causam danos à máquina, mas que, diferente dos *Spywares*, visam trazer telas de propagandas, geralmente desagradáveis.

Assim, vê-se que existe uma gama enorme de ameaças para os computadores, sendo a função do Anti-vírus muito importante na segurança de computadores e na manutenção dos mesmos.

2.2. BREVE HISTÓRIA DOS ANTI-VÍRUS

Dada a importância dos anti-vírus supracitada, primeiramente daremos uma breve explanação histórica dos mesmos, para depois explicarmos melhor o seu funcionamento.

A história da criação dos vírus e anti-vírus é muito controversa. Muitos autores atribuem o título de criação desses termos a pessoas diferentes, sendo também muito difícil distinguir objetivamente quando um programa se tornou mesmo um vírus e quando não.

Mas se analisarmos do ponto de vista cronológico, podemos atribuir estes títulos a um americano denominado Fred Cohen, que se não pode ser considerado o criador, é no mínimo considerado o pai do anti-vírus, sendo inegável sua contribuição no assunto.

Em 1983, quando ainda era estudante de engenharia na *University of Southern California*, ele escreveu um programa parasita que consumia todo o processamento da CPU do computador, constituído este o primeiro vírus ou o precursor deste. Conta-se que ele o fez na aula do famoso Leonard Adleman, um dos três cientistas famosos pela criação do primeiro sistema de criptografia assimétrica, o RSA.

Mais tarde, Fred Cohen escreveu vários artigos sobre os vírus de computadores. Em um deles, ele propôs a primeira técnica de detecção viral, a de verificação de assinaturas virais em arquivos.

O funcionamento da técnica é simples. Todo o vírus ou *malware* possui um conjunto de instruções necessárias para seu funcionamento. Esse conjunto de instruções, geralmente em Assembly, gera um conjunto de *bytes* que são armazenados ou num arquivo próprio pra ele, ou em um arquivo infectado.

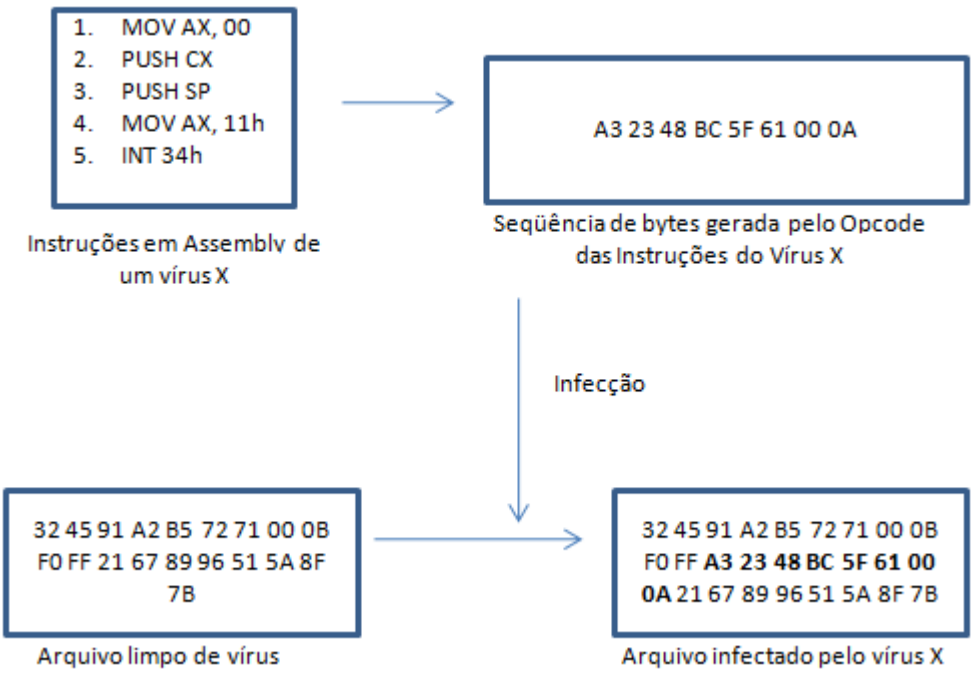

FIG. 2-1: Assinatura viral e a sua presença em arquivos infectados

Conhecido esse conjunto de *bytes*, é possível para um anti-vírus detectar a presença de um vírus fazendo uma varredura em todos os arquivos do sistema, procurando por esta seqüência de *bytes* em cada arquivo.

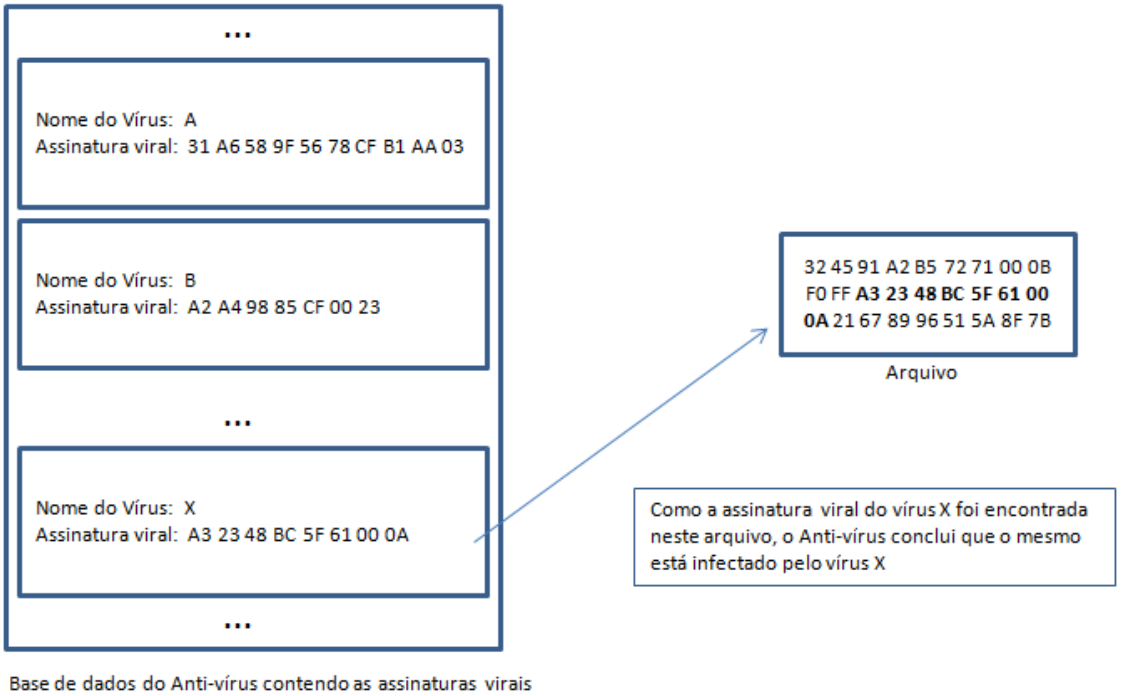

FIG. 2-2: Esquema de detecção de vírus por meio de checagem de assinatura viral

Embora seja bastante antiga, a verificação de assinaturas virais (*signature scanning*) é ainda a melhor técnica de detecção viral, sendo bastante usada nos dias de hoje. As outras técnicas são utilizadas apenas para ajuste fino na detecção.

O renomado livro "*Handbook of Security Information*" (BIDGOLI 2006) faz uma declaração sobre o assunto na página 451: "*Signature scanning is the most common, accurate, and effective technique currently used in the fight against malicious software. It is the method employed by the earliest antivirus software and it continues to be an important piece of all comprehensive antivirus products.* (...)".

2.3. O PROBLEMA DA INDECIBILIDADE

Quando descobrimos que nosso computador está infectado e possuímos um anti-vírus instalado na maioria das vezes temos a seguinte reação: ou atribuímos a causa da infecção a falta de atualização do anti-vírus, ou então questionamos a eficiência do mesmo.

E quando estamos com um anti-vírus instalado e atualizado, por pior ainda, tendemos a superestimar a eficiência do mesmo, negligenciando determinados procedimentos de segurança, como o cuidado em *sites* que se entra na *internet.*

Apesar de muitas pessoas desconhecerem esse fato, por mais atualizada que a base de dados do mesmo esteja, um anti-vírus nunca ter como sempre detectar a presença de ameaças novas em um computador.

Na verdade, foi provado que é **matematicamente impossível** que um anti-vírus tenha 100% de precisão na detecção viral. Em 1987, através de seu famoso artigo "*Computer Viruses Theory and Experiments"* (COHEN 1987), se valendo da teoria da Máquina Universal de Turing, Fred Cohen prova que o problema da detecção exata de um vírus é um problema **indecidível**, i.e., não-computável, que não pode ser implementado por um programa de computador. Esta foi a maior contribuição dele feita no ramo de Anti-vírus da Computação.

Primeiramente, define-se o que é um vírus em termos de linguagem, valendo-se da peculiaridade da auto-replicação dos mesmos.

Em seguida, supõe a existência de uma máquina de Turing para reconhecer a linguagem que compõe todos os vírus.

Finalmente, mostra-se que, com o auxílio dessa máquina de Turing que reconhece a linguagem dos vírus, é possível também construir uma máquina de Turing capaz de reconhecer a linguagem relativa ao clássico Problema da Parada (*The Halting Problem*), que

pode ser descrito como o problema de um programa determinar se outro programa pára, não ficando em "*loop* infinito". Ou seja, reduz-se o problema do reconhecimento da linguagem viral ao Problema da Parada.

Assim, se a máquina de Turing que reconhece a linguagem dos vírus existe, então a máquina de Turing para o Problema da Parada também existe. Como Alan Turing provou em 1936 que o Problema da Parada é indecidível, então não existe Máquina de Turing para o Problema da Parada. Conseqüentemente, não existe Máquina de Turing para reconhecer a linguagem viral e, dessa maneira, esse problema também é indecidível (ou não-computável).

Para mais detalhes, consulte "*Turing Machines and Undecidability with Special Focus on Computer Viruses*" (ANDERSSON 2003), que dá uma excelente explicação sobre o assunto, mostrando passo a passo como Fred Cohen provou a indecidibilidade deste problema.

Outra forma de mostrarmos mais informalmente que é impossível para um anti-vírus possuir 100% de precisão na detecção viral é através da técnica da contradição.

Suponhamos que exista um algoritmo A(P), tendo como entrada um programa P, que retorne "Sim" sempre que o programa P seja um vírus e "Não" sempre que ele não seja. Munido deste algoritmo, um vírus V pode usar este algoritmo e modificar o seu código até que A(V) retorne "Não". Assim, temos um vírus V que segundo o algoritmo A não é um vírus, o que é obviamente uma contradição lógica.

Essas duas explanações sobre a impossibilidade de detecção exata de um vírus ésão importantíssimas para a perfeita compreensão do funcionamento do anti-vírus.

Se é impossível detectar com 100% de precisão um vírus, então concluímos que muitas vezes esses algoritmos de detecção viral, destacando-se entre eles o de assinatura viral, sempre erram ou pra menos ou pra mais, i.e., ou esses algoritmos falham na detecção de um vírus, ou então detectam um chamado Falso Positivo, um arquivo que é acusado como infectado de vírus quando realmente não está.

Assim, a evolução das técnicas de detecção viral tem sido realizada no intuito de reduzir o número de ocorrência de falsos positivos e de falhas em detecções virais.

2.4. EVOLUÇÃO DAS TÉCNICAS DE DETECÇÃO VIRAL

Na seção anterior, vimos que a técnica de detecção viral por checagem de assinatura viral nem sempre é precisa e os anti-vírus tem evoluído bastante para reduzir as falhas de detecção e a detecção de falsos positivos.

É óbvio que essa evolução tem sido fomentada pela batalha travada entre os desenvolvedores de vírus e os de anti-vírus. Da mesma maneira, como ao longo da História da Humanidade podemos notar que a evolução das técnicas de criptografia tem sido estimulada pelo trabalho dos criptoanalistas, o desenvolvimento dos anti-vírus têm sido estimulados pelo desenvolvimento dos vírus, que se tornam cada vez mais sofisticados para responder às técnicas de detecção dos mesmos.

Nesta seção, analisaremos a evolução dos anti-vírus ao longo da história pelo menos até o que se tem conhecimento hoje.

Inicialmente, na década de 80, quando os primeiros vírus foram criados, estes eram simples programas que infectavam um arquivo executável, adicionando a estes rotinas que davam ao vírus controle do sistema e que lhe permitiam infectar outro arquivo.

Para o anti-vírus, era simples a detecção destes vírus, bastando que o anti-vírus descobrisse a sequência de *bytes* que representavam as instruções do vírus (assinatura viral) e checar a presença delas em arquivos.

Em resposta, os desenvolvedores de vírus criaram o conceito de vírus encriptado. Ao invés do vírus infectar um arquivo com suas instruções diretamente, ele primeiro as encriptava antes de anexá-las no arquivo. Quando um vírus encriptado se encontrava num arquivo, ele primeiro tomava controle do sistema, depois executava algumas rotinas de decriptação nas suas instruções de infecção de outros arquivos e infectava estes arquivos injetando neles instruções criptografadas, cuja chave era mudada de infecção em infecção. Dessa forma, a assinatura viral não se mantinha constante, impedindo o *scanner* do anti-vírus de lograr êxito em sua busca por vírus.

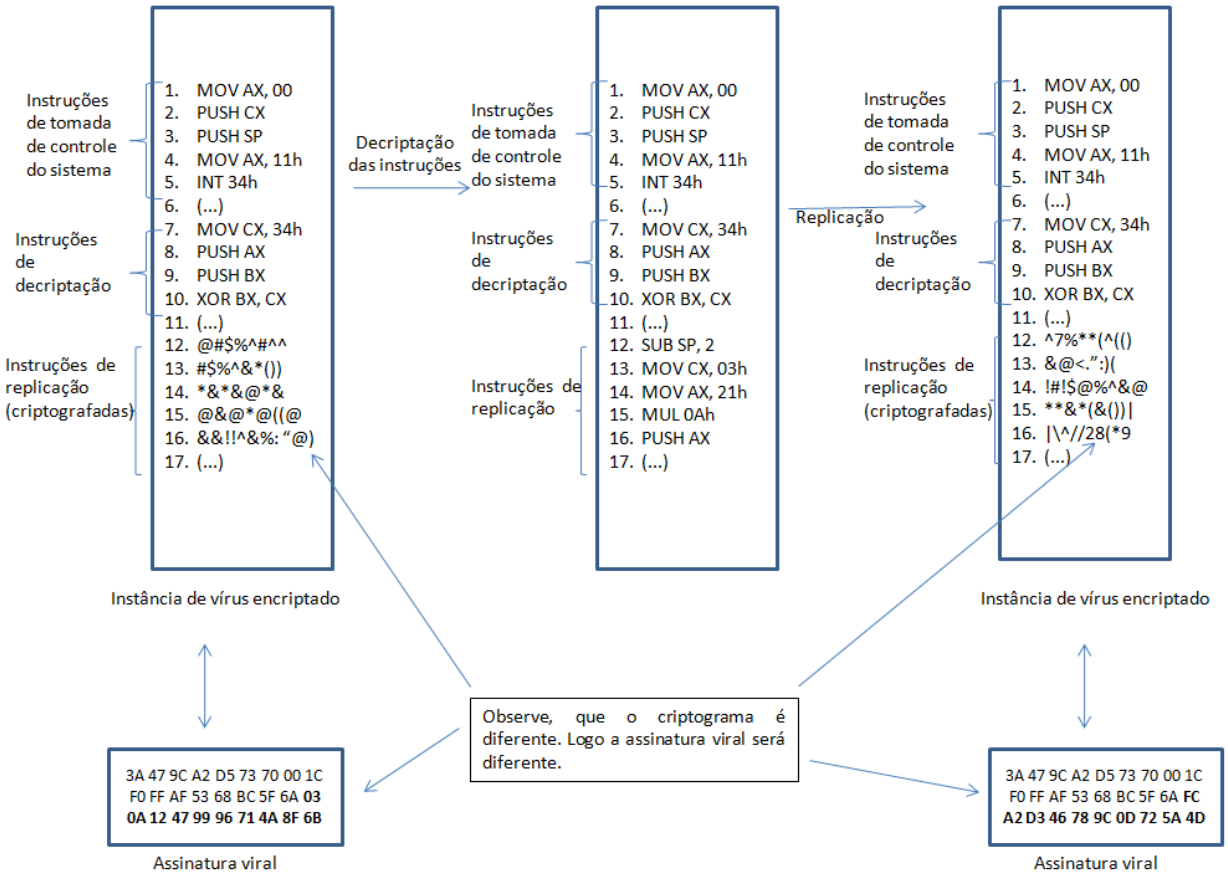

FIG. 2-3: Propagação de um vírus encriptado

Apesar de parecer a primeira vista bastante eficiente este método possui uma vulnerabilidade: as instruções de tomada de controle do sistema e de decriptação precisavam permanecer descriptografadas. Conseqüentemente, essas instruções geravam uma seqüência de *bytes* constante e poderia ser usada como assinatura viral, ao invés de se usar todo o conjunto de instruções do vírus.

Assim, a base de dados dos anti-vírus foram reconfiguradas para conter somente essas instruções como assinaturas virais e, deste modo, o problema dos vírus encriptados foi resolvido.

Em retaliação, os desenvolvedores de vírus criaram os famosos vírus polimórficos, por volta do começo da década de 90, que continham também uma *engine* de mutação que alterava aleatoriamente também as rotinas de tomada de controle do sistema e decriptação dos vírus encriptados, tornando a assinatura viral desses vírus inúteis.

Podemos destacar para descrever o funcionamento dessas *engines* dois princípios.

O primeiro constitui em trocar instruções em Assembly por outras equivalentes. Por exemplo, na figura notamos que podemos zerar um registrador de diferentes formas. Cada uma dessa forma gera uma seqüência de *bytes* diferentes e assim uma assinatura viral diferente.

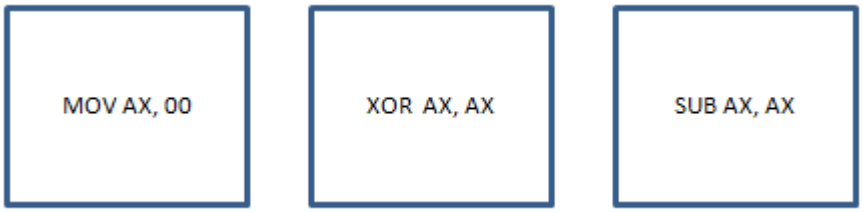

FiG. 2-4: Diferentes maneiras de se zerar um registrador

O segundo constitui em adicionar instruções em Assembly aleatórias que não fazem nada de útil. Isto permite que na sequência de *bytes* haja um trecho aleatório, que inviabilize a detecção de padrões para comporem uma assinatura viral.[2]

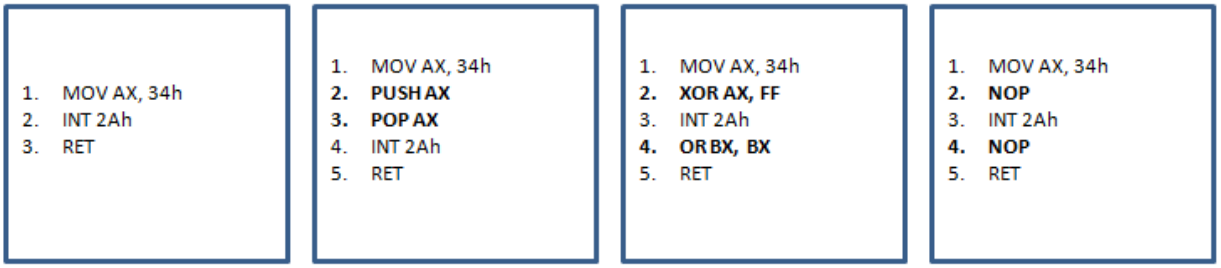

FIG. 2-5: Diversas maneiras de inserir instruções irrelevantes

[2] A inserção de trechos aleatórios sem valor é também uma técnica muito utilizada em criptografia e nela é denominada *salting*.

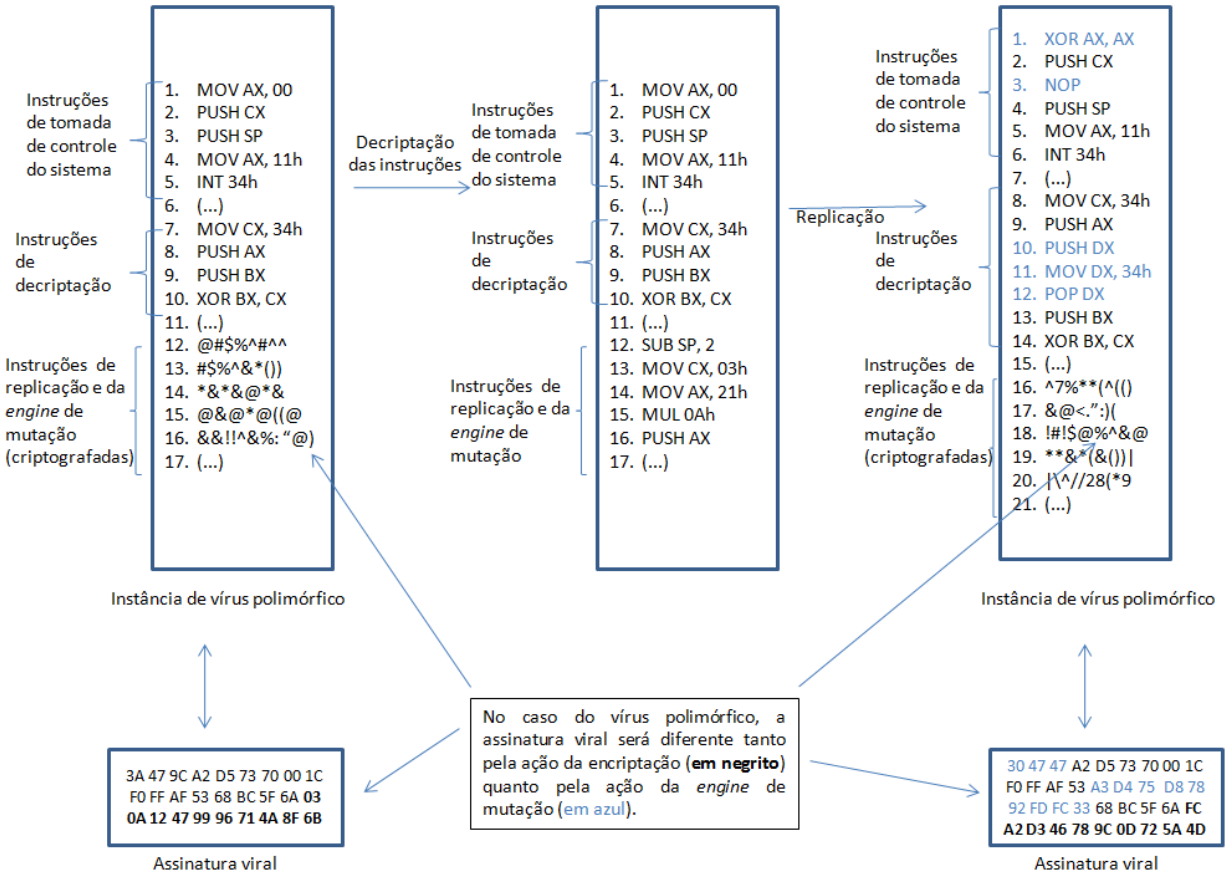

FIG. 2-6: Propagação de um vírus polimórfico

Em 1991, os famosos vírus *Tequila* e *Maltese Amoeba*, primeiros de sua raça, causaram largas infecções.

Em 1992, um dos seus autores, denominado *Dark Avenger*, disponibilizou a outros *crackers* o código fonte dessa *engine* de mutação e instruções de como utilizá-la para a construção de seus próprios kits de vírus polimórficos. Dessa forma, várias pragas polimórifcas surgiram nesse período.

Para piorar a situação, tornou-se moda entre os *crackers* trocarem entre si código fonte dessas *engines*, tornando a variação desses vírus ainda maior.

Esses vírus deram aos desenvolvedores de anti-vírus muita dor de cabeça e várias noites mal dormidas. Durante muito tempo, esse tipo de vírus ficou sem solução de detecção, sendo por algum tempo implementado técnicas de captura de vírus polimórficos específicas para cada vírus, o que ao longo do tempo, com o crescimento dos mesmos, tornou-se impraticável e muito custoso.

Finalmente, os anti-vírus obtiveram êxito na captura de vírus polimórficos com o advento das chamadas Técnicas de Heurísitica. Esse conjunto de técnicas se baseia em na procura de

comportamentos inconsistentes realizado pelas instruções do arquivo que se suspeita de infecção.

Um exemplo de heurística é o descrito pela figura abaixo. Inicialmente tem-se uma probabilidade fixa de o arquivo está infectado (digamos 10%, por exemplo). A cada regra a favor da infecção, aumenta-se a probabilidade de infecção em um determinado valor. A cada regra contra a infecção, diminui-se a probabilidade de infecção em um determinado valor.

Regras	Alteração na Probabilidade
Encontrar um instrução NOP	+1%
Verificar que o conteúdo de um registrador é removido, logo após ser carregado.	+0,5%
Encontrar uma chamada de interrupção do DOS	-15%
A cada 100 instruções, não encontrar nenhuma escrita em memória.	-5%

TAB. 2-1: Exemplo de regras de Heurística

Quando chega-se numa determinada taxa de probabilidade, não necessariamente 100%, assume-se que o arquivo está infectado.

Esse tipo de técnica que permite a análise comportamental de um arquivo para avaliar se este encontra-se infectado, resolveu de forma bastante eficiente o problema dos vírus polimórficos.

É claro que os desenvolvedores de vírus certamente criaram outras técnicas, que ainda não são de conhecimento público, que burlam as técnicas de heurística de um anti-vírus. Essa batalha entre vírus e anti-vírus não terminou e nunca há de terminar, visto que é impossível o anti-vírus obtenha precisão de 100% em detecções virais.

Todavia, essa breve explanação a cerca das técnicas de detecção viral nos pode dar uma idéia de quão complexo é o desenvolvimento das mesmas, mesmo que a idéia das técnicas seja ou pareça bastante simples.

2.5. ANTI-VÍRUS ESTÁTICO E DINÂMICO

Com o advento das técnicas de heurística para enfrentar os vírus polimórficos, surgiu uma nova modalidade de anti-vírus: o anti-vírus dinâmico.

Tecnicamente, um anti-vírus estático é aquele que utiliza técnicas de detecção viral estático, i.e., técnicas que levam em conta o arquivo possivelmente infectado isoladamente, analisando como é a sua estrutura interna, se ele possui determinadas características que se assemelham a um arquivo contaminado.

São diversas as técnicas que podem ser usadas para analisar o arquivo isoladamente. A principal já foi explicada anteriormente e é a análise de assinaturas. Outra técnica muito empregada é a verificação dos *entry points*, os pontos de entrada de arquivos executáveis, cuja localização geralmente é modificada pelos vírus, visando adicionar a sua funcionalidade ao arquivo executável.

Já o anti-vírus dinâmico, além das técnicas de análise estática, também se vale das técnicas de analise dinâmica. Esse tipo de análise procura monitorar o comportamento do arquivo infectado no ambiente do sistema operacional ao qual ele se insere. Por exemplo, se tal arquivo, geralmente executável, costuma acessar muito o registro do Windows e o modificar, então ele é um arquivo suspeito de estar infectado.

Hoje em dia, tais técnicas de análise dinâmica são imprecidíveis para a captura dos vírus polimórficos, que atualmente compõe um parcela considerável das pragas digitais existentes.

Apesar disso, conforme foi dito na descrição dos objetivos do projeto fim de curso, neste trabalho não fará parte do escopo a inclusão de técnicas de análise dinâmica, se preocupando exclusivamente em manter o anti-vírus estático, utilizando especificamente a técnica de detecção por meio de checagem de assinatura viral, descartando mesmo o uso de outras técnicas estáticas como a análise dos *entry points* nos PEs[3] .

2.6. VERIFICAÇÃO ON-DEMAND E ON-ACCESS

Até agora falamos das técnicas de checagem de assinaturas virais e da sua influência na captura dos diferentes tipos de vírus que existem. Mas em nenhum momento, discutimos a eficiência das mesmas em termos de custo de tempo e de recursos do sistema.

É óbvio que a varredura de todos os arquivos de um sistema operacional e a análise *byte* a *byte* para detecção de seqüências de *bytes* que correspondam a assinaturas é um procedimento extremamente dispendioso.

[3] *Portable Executable*: é o nome dado aos executáveis de 32-bits ou 64-bits do sistema operacional Windows

Pensando nisso, os desenvolvedores de anti-vírus criaram um novo conceito em termos de verificação de assinaturas: o de verificação *on-access*[4]. Ao invés de varrer todos os arquivos do sistema operacional em busca de assinaturas, varre-se automaticamente apenas os arquivos que são acessados pelo sistema operacional, reduzindo bastante a carga de processamento do anti-vírus e tornando o *scanner* mais inteligente e mais transparente ao usuário.

No entanto, a varredura de todos os arquivos do sistema operacional não foi descartado, sendo essa funcionalidade mantida para caso o usuário deseje manualmente escanear todos os arquivos. A essa funcionalidade foi dado o nome de verificação *on-demand* (sob demanda).

Através dessa *feature* de verificação *on-access*, quando um arquivo é acessado no sistema operacional, o anti-vírus efetua antes uma verificação nesse arquivo, de modo a determinar se ele está infectado ou não. Se ele estiver infectado, o anti-vírus avisa o usuário para que este decida se o acesso ao arquivo deve ser permitido ou se deve ser vetado. Dessa forma, o sistema operacional do usuário é protegido sem que o usuário precise agir de forma ativa junto ao anti-vírus para iniciar uma verificação de arquivos.

De um ponto de vista técnico, a verificação *on-access* requer a interceptação de algumas *system calls* de acesso ao *file system*. Dessa forma, qualquer solução que se proponha a implementar verificação *on-access* em vários sistemas operacionais precisará necessariamente de uma implementação diferente e dependente de plataforma para cada sistema operacional.

Uma proposta de implementação, que visava contornar esse problema de dependência do sistema operacional, é o projeto *Dazuko*. Este tem por finalidade criar uma camada de *software* entre as aplicações que desejam ser notificadas de um acesso ao *file system* e o sistema operacional subjacente. O Dazuko então fica responsável pela parte de interceptar as chamadas de acesso a arquivos e notificar as aplicações interessadas. Com isso, uma aplicação de anti-vírus não precisaria se preocupar com detalhes específicos de cada sistema operacional, apenas com a lógica de *scanning* de arquivos.

[4] Tal funcionalidade é associada também ao conceito de *Residência*. Todo o anti-vírus chamado de *residente*, possui por definição a funcionalidade de verificação *on-access*.

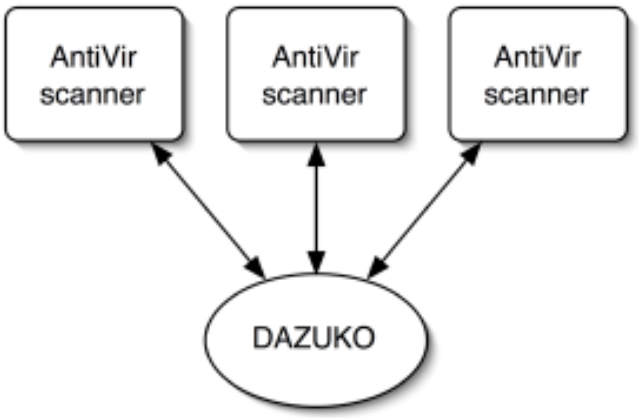

FIG. 2-7: Vários anti-vírus podem usufruir da abstração fornecida pelo Dazuko

Segundo o artigo "*Dazuko: An Open Solution to Facilitate 'On-Access' Scanning.*" (OGNESS 2003), o projeto Dazuko apresenta uma arquitetura em três camadas que possibilita facilmente a integração com qualquer sistema operacional existente. A arquitetura está representada na figura a seguir:

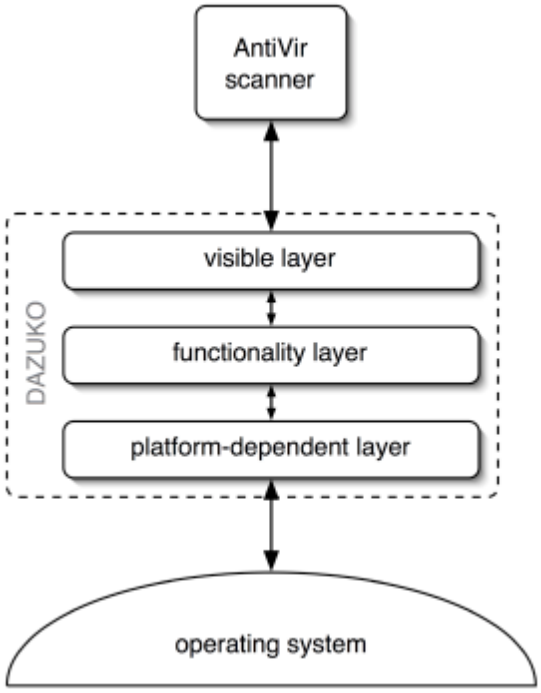

FIG. 2-8: Arquitetura em três camadas do Dazuko

A camada mais inferior, *platform-dependent layer*, é a responsável por interagir com o sistema operacional e interceptar as *system calls* relativos ao acesso ao *file system* e passar as informações relacionadas a elas para a camada intermediária, *functionality layer*, que realizará

efetivamente o processamento das requisições de acesso. A camada mais superior, *visible layer*, expõe para as diversas aplicações do sistema uma interface através da qual elas podem se cadastrar para receber notificações, provenientes da *functionality layer*, cada vez que ym acesso ao *file system* é feito.

O Dazuko foi implementado inicialmente apenas para o Linux. Em teoria, apenas a camada mais inferior precisaria ser reescrita para que o Dazuko fosse portado para outros sistemas operacionais. Na prática, o código do projeto está muito particularizado para a plataforma Linux e, pior ainda, muito mal-documentado também. Dessa forma, torna-se muito difícil utilizá-lo para implementar *on-access scanning* no Windows.

A solução encontrada para implementar essa funcionalidade no Windows foi a utilização dos chamados *File System Filter Drivers*, que serão abordados melhor posteriormente no capítulo relacionado a *drivers* do Windows.

3. CLAMAV

3.1. DESCRIÇÃO

No capitulo anterior, foi apresentado a figura do anti-vírus, tendo-se uma idéia de como ele funciona, suas principais funcionalidades e como foi sua evolução ao longo dessas duas décadas que se passaram.

Existem diversos anti-vírus disponíveis no mercado como o Karpesky, Norton, AVG, Avast, McAffe, Norman, etc. No entanto, a maioria deles não é *free* ou então possui uma versão gratuita, mas essa é *Trial*, i.e., depois de um tempo expira e para restaurá-lo é necessário comprar uma licença. Poucos são os anti-vírus verdadeiramente livres, que não são *trials*, e possibilizam atualizações constantes e sem limite de vezes.

O principal anti-vírus verdadeiramente livre no mercado é o **Clam Antivirus**, mas conhecido como **ClamAv**. Trata-se de um anti-vírus desenvolvido para Linux/Unix, projetado para ser um *scanner* de análise estática de assinaturas virais, cujo a *engine* é disponibilizada na forma de biblioteca, denominada **LibClamAv**.

FIG. 3-1: Ícone do ClamAV

Historicamente, o ClamAv foi criado mais ou menos em janeiro de 2002. É licenciado pela GPL e mantido por sua comunidade[5], juntamente com as atualizações de suas versões e de sua base de dados.

Desde sua criação, tem sido bastante alvo de elogios pela comunidade livre e também por diversas outras comunidades, que mesmo não sendo livres, elogiam bastante a iniciativa de construção de anti-vírus livre, para romper com a supremacia das empresas vendedoras de anti-vírus que mantém a todo o custo seu código fechado.

Em 2005, o site *Information Week*[6] considerou sua base de dados a melhor do mundo em matéria de anti-vírus, devido a constante atualização da mesma, que justamente por ser livre, recebe uma imensa contribuição de diversos desenvolvedores.

[5] Para maiores detalhes, acesse: www.clamav.net

Em 2007, a revista *Linux World* realizou um teste confrontando 10 anti-vírus com 25 anti-vírus e somente três deles conseguiram detectar todos os vírus: o ClamAv, o Kapersky e o Symantec.

Desde sua criação, ele tem sido largamente utilizado em seu propósito inicial: oferecer segurança para máquinas Linux/Unix. Como os vírus existentes para esses sistemas operacionais são pequenos, a base de dados acabou agregando também os vírus para Windows, de modo que ele também passou a ser utilizado no escaneamento de partições NTFS e FAT32, em máquinas com *boot* dual, tanto com Windows quanto Linux.

Segundo seu site principal, quando rodado diretamente no Linux, ele possui as seguintes funcionalidades:

- *Scanner* de linha de comando;
- *Daemon*[7] *multi-thread* que suporta verificação *on-access*;
- Atualizador automático da base de dados;
- Interface *milter* para o *SendMail*;
- Biblioteca C para análise viral;
- Múltiplas atualizações diárias no site do ClamAv;
- Suporte incluso para vários formatos de arquivos, como *Zip*, *RAR*, *Tar*, *Gzip*, *Bzip2*, *OLE2*, *Cabinet*, *CHM*, *BinHex*, *SIS*, etc;
- Suporte incluso para vários formatos de e-mails;
- Suporte incluso a *ELF*, PEs comprimidos com *UPX*, *FSG*, *Petite*, *NsPack*, *wwpack32*, *MEW*, *Upack* e ocultados com *SUE*, *Y0da Cryptor*, entre outros;
- Suporte incluso a formatos de documentos populares como MS Office, MacOffice, HTML, RTF e PDF;

Como se vê, o ClamAv é bastante desenvolvido em matéria de anti-vírus para o Linux/Unix. A seguir será mostrado como é sua arquitetura nesse sistema operacional.

---

[6] Para maiores detalhes, consulte: http://www.informationweek.com/news/software/linux/showArticle.jhtml?articleID=166400446

[7] *Daemon* é como são chamados os programas que rodam em segundo plano nos Linux e Unix, sem intervenção alguma do usuário. No Windows, esses programas são chamados de **Serviços**.

3.2. ARQUITETURA DO CLAMAV NO LINUX

Quando se instala o ClamAv no Linux a princípio o usuário acostumado com manuseio de anti-vírus no ambiente Windows pode criar uma expectativa de que após a instalação deste antivírus, ele venha a se deparar com uma interface gráfica bonita e um programa único sem subdivisões em componentes. Essa está longe de ser a realidade.

O ClamAv em si não possui nenhuma interface gráfica com o usuário[8], sendo inteiramente operado por linha de comando (até para manter a tradição dos programas escritos em Linux).

E para facilitar a manutenção e configuração, ele também é dividido em diversos componentes, cada um com uma funcionalidade específica. São eles: *clamscan*, *clamd*, *clamuko*, *freshclam*, *sigtool* e *libclamav*.

O **clamscan** (*ClamAv Scanner*) é o componente do ClamAv responsável pelo escaneamento *on-demand* do arquivos via linha de comando.

O **clamd** (*ClamAv Daemon*) se responsabiliza por todo o escaneamento *on-access* no computador. Ele que utiliza *LibClamAv* para escanear os arquivos vindo diretamente do *clamuko*.

O **clamuko** (*ClamAv + Dazuko*) é o componente do ClamAv que faz a interface com o *Dazuko* e, deste modo, juntamente com o clamd desempenha o papel de escaneamento *on-access*.

Conforme supracitado no capítulo anterior, uma das maneiras de se implementar o escaneamento *on-access* em um sistema operacional é utilizando o projeto *Dazuko*, desde que o sistema operacional em questão tenha suporte ao mesmo. O Linux possui suporte ao *Dazuko*, desde que o módulo do Kernel *Dazuko* esteja instalado e carregado no mesmo.

Assim, enquanto o *clamd* se preocupa em escanear o arquivo em questão que está sendo acessado por algum programa, através das bibliotecas de verificação de assinatura viral, o *clamuko* faz a interface com o *dazuko* (especificamente com a *visible layer* desse), para receber os arquivos que estão sendo acessado pelos programas e, assim, passá-los ao *clamd* para serem escaneados.

O **freshclam** é um componente importantíssimo, sendo responsável pela atualização da base de dados do ClamAv. É ele que se conecta ao site do ClamAv ou aos seus *mirrors* e

[8] Na verdade, existem diversos projetos de *front-end* gráficos para o ClamAv.
O mais famoso é o KlamAv um *front-end* para o KDE. Para mais detalhes a cerca do mesmo, consulte: http://klamav.sourceforge.net/klamavwiki/index.php/Main_Page

baixa os arquivos novos contendo novas assinaturas de vírus ou excluindo assinaturas antigas que comprovadamente tenham sido concluídas que se tratam de falsos positivos.

A estrutura da base de dados do ClamAv é composta de dois arquivos de formato CVD (*ClamAv Virus Database*), o **main.cvd** e o **daily.cvd**. O primeiro arquivo, o maior, possui as principais assinaturas virais da base de dados. Já o segundo contém apenas as assinaturas virais que são atualizadas diariamente.

Cada arquivo desse é composto de um cabeçalho de 512 *bytes* e de uma seqüência de uma ou várias estruturas de dados `cl_cvd` representando as assinaturas virais, sendo o arquivo comprimido utilizando *tarball*.

Cada estrutura de dados dessa é constituída de uma seqüência de strings contendo informações como o nome do *malware*, o tipo de *malware* (vírus, *trojan*, *worm*, etc), a assinatura propriamente dita em hexadecimal, uma *checksum* em MD5 para checar a integridade dessa assinatura digital e informações de quando essa assinatura foi criada ou alterada pela última vez.

Na seqüência, temos o componente **sigtool** (*signature tool*). Trata-se de uma ferramenta para edição e criação de assinaturas virais a partir de arquivos executáveis. Essas assinaturas criadas são armazenadas em arquivos HDB, não sendo incorporadas diretamente a base de dados, mas devendo serem enviadas ao site do ClamAv para posterior averiguação dos desenvolvedores[9].

Por último, temos o componente mais precioso do ClamAv: a sua *engine* em forma de biblioteca C denominada **LibClamAv** (*ClamAv Library*). A LibClamAv contém todas as funções em C necessárias para o escaneamento de arquivos, identificação dos vírus que estão contaminando-os e também as funções que permitem atualizar a base de dados e recarregar em tempo de execução na memória principal a base de dados recém-atualizada.

A seguir nas próximas seções serão descritos como se utliza o **clamscan** e **freshclam** e como é a estrutura interna da API da **LibClamAv**. O freshclam e a LibClamAv são importantíssimos no desenvolvimento do anti-vírus, conforme será observado nos próximos capítulos, principalmente no Capítulo referente a arquitetura do ClamEB.

[9] Para maiores detalhes a cerca do envio de assinaturas virais, consulte: http://cgi.clamav.net/sendvirus.cgi

3.3. CLAMSCAN

Apesar deste componente não ser utilizado no protótipo do anti-vírus, a utilização do mesmo no Linux foi incluída nesta monografia para se ter uma idéia clara de como ele funciona. Além disso, para efeitos de desenvolvimento do protótipo, a compreensão deste componente foi um passo intermediário importantíssimo.

Como dito anteriormente, este é um *scanner* de linha de comando. Assim para, por exemplo, se escanear o diretório corrente no Linux, é necessário digitar o comando:

```
clamscan --stout
```

Esse comando escaneará todos os arquivos contidos no diretório corrente mostrando na tela os resultados desse escaneamento. Um exemplo de saída para este comando é:

```
/home/jose/arquivo1.exe: Worm.Sober FOUND
/home/jose/arquivo2.o: OK
/home/jose/arquivo3.c: OK
/home/jose/arquivo4.c: OK
/home/jose/arquivo4.hta: VBS.Inor.D FOUND
```

No entanto, este comando não escaneará os subdiretórios presente neste diretório corrente. Para fazê-lo, coloque a opção `-r` ou `--recursive`, como segue o exemplo a seguir:

```
clamscan –r --stout
```

Para escanear um outro diretório que não seja o diretório corrente ou um arquivo específico, coloque como argumento esse diretório ou arquivo após a inclusão das opções, conforme segue o seguinte exemplo:

```
clamscan –r --stout /home/maria/
```

Até agora, esses comandos assumiram que a base de dados a ser utilizada é a que se encontra no diretório padrão assumido durante a instalação do ClamAv. Para fazer com que a base de dados utilizada seja outra localizada em outro diretório diferente utlilize a opção `--database=ARQUIVO_OU_DIRETORIO`, conforme o exemplo a seguir:

```
clamscan –-database=/home/jose/cvd/ –r --stout /home/maria/
```

Existem diversas outras opções mais complexas. Uma delas é a que restringe o escaneamento a arquivos menores que um determinado tamanho. Ela feita com a opção `--max-space=TAMANHO`, conforme o exemplo a seguir que restringe o escaneamento a arquivo menores que 100 *megabytes*.

```
clamscan --max-space=100m –r --stout /home/maria/
```

Existem outras opções que, por exemplo, desabilitam o uso de alguns algoritmos da bilbioteca do ClamAv. Para maiores detalhes, consulte a documentação do ClamAv ou digite o comando no Linux:

```
man clamscan
```

3.4. FRESHCLAM

Conforme dito anteriormente, o freshclam é o componente responsável pela atualização da base de dados do ClamAv. O processo de atualização como um todo, constituído da verificação se o banco de dados realmente está desatualizado, da conexão com algum *mirror*, do *download* dos arquivos cvd da substituição dos antigos, é completamente transparente ao usuário.

A compreensão de como se o utiliza é fundamental, pois é através do mesmo que, conforme será visto mais tarde, a atualização do ClamEB será feita.

O freshclam é basicamente rodado em dois modos:

- **Modo interativo**: em que o usuário o ativa via linha de comando e o freshclam retorna aos poucos na tela os passos que costumam ser executados;

- **Modo *daemon***: em que o mesmo funciona como um *daemon*, ou seja, roda em segundo plano, realizando as atualizações silenciosamente e sem qualquer intervenção do usuário.

Para funcionar corretamente, é necessário que o freshclam possua no seu diretório um arquivo ou em outro caso seja especificado um arquivo de configuração denominado **freshclam.conf**. Nesse arquivo é armazenado basicamente o endereço onde se encontram os *mirrors* para *download* dos arquivos CVD e compõe a base de dados, o servidor de DNS com a resolução de nomes para esses *mirrors* e o número de tentativas que serão feitas para se conectar a um desses *mirrors*.

Para executar manualmente a atualização de dados, digite:

```
freshclam
```

Esse comando fará o *download* da nova base de dados no diretório corrente substituindo os arquivos antigos caso eles existam. Esse comando também leva em conta que o arquivo `freshclam.conf` está localizado no diretório corrente, resultando em erro caso não esteja. Caso o arquivo de configuração esteja em outro local, pode-se utilizar a opção `--config-file=ARQUIVO`.

```
freshclam --config-file=ARQUIVO
```

Para informar ao freshclam que a base de dados se encontra ou deverá ser baixada em outro diretório que não seja o corrente, utilize a opção `--datadir=DIRETORIO`.

```
freshclam --datadir=DIRETORIO
```

Para forçar o freshclam a não utilizar a resolução de nomes (DNS) utilize a opção `--nodns`. Esta opção pode ser útil caso haja algum problema com servidores de DNS ou caso deseje-se reduzir o tempo de operação do freshclam.

```
freshclam --nodns
```

Para executar o freshclam em modo *daemon*, digite o comando abaixo no Linux, sendo muito provável dele já estar ativo desde o *boot*.

```
freshclam -d
```

Para fazer com que o freshclam, ao rodar em modo *daemon*, faça a atualização da base de dados um numero N de vezes ao dia, digite:

```
freshclam –d –c N
```

As opções `--on-error-execute=COMANDO`, `--on-update-execute=COMANDO` e `--on-outdated-execute=COMANDO` também podem ser útil para se executar algum programa caso haja algum problema com o freshclam como respectivamente erro na atualização, sucesso na mesma ou a base esteja desatualizada.

```
freshclam –d –c 3 --on-error-execute=COMANDO
freshclam –d –c 3 --on-update-execute=COMANDO
freshclam –d –c 3 --on-outdated-execute=COMANDO
```

Embora não seja útil quando se executa em linha de comando o freshclam, é importante termos uma consciência das saídas que ele retorna, uma vez que, quando se executa o mesmo dentro de outro programa, como é o caso no ClamEB, conforme será visto mais tarde, tais valores de retorno do mesmo são importantíssimos.

A tabela abaixo mostra os principais valores de retorno como saída do freshclam e o seus respectivos significados:

Valor de retorno da saída	Significado
0	A base de dados foi atualizada com sucesso.
1	A base de dados não estava desatualizada, não sendo ncessário atualizá-la novamente.
52	Problemas de rede ou de conexão.
54	Erro durante a verificação da soma MD5 nos arquivos CVD.
56	Erro no arquivo de configuração.
58	Erro devido a impossibilidade de se ler a base de dados em um servidor remoto.
59	Erro devido aos *Mirrors* não estarem sincronizados no momento (deve-se então tentar mais tarde).
61	Erro devido à falta de privilégios.
62	Erro devido a não se poder inicializar o *logger*.

TAB. 3-1: Principais valores de retorno da saída do freshclam e seus significados

3.5. LIBCLAMAV

Sem dúvida nenhuma a característica mais peculiar do ClamAv sobre todos os outros anti-vírus não é o fato dele ser somente livre, mas o fato de toda a sua *engine* ser disponível em forma de biblioteca C.

Em outras palavras, é a LibClamAv que torna esse anti-vírus tão especial, porque é através dessa *engine* em forma de biblioteca que se pode construir mais facilmente outros anti-vírus como este protótipo a que se destina esse projeto final de curso.

Para ser utilizada, a LibClamAv conta um conjunto de funções denominado API (*Application Programming Interface*), que são as únicas funções e estruturas de dados que deverão ser utilizadas pelo usuário.

Esse conjunto de funções e estruturas são reunidas em uma arquivo de *Header C*[10] denominado **clamav.h**, que foi escrito para ser incluso no programa C que o utiliza. Assim, o primeiro passo para utilizar a LibClamAv é incluir esse *header* no corpo do seu programa da seguinte maneira:

```
#include <clamav.h>
```

[10] Em C, basicamente, existem dois tipos de arquivos: os ".h" e os ".c". Os primeiros são os *hedaers*, ou seja, os arquivos contendo apenas as declarações de cabeçalhos de funções, *typedefs*, etc. Já os segundos são aqueles que contem a implementação desses cabeçalhos de funções.

Se analisarmos com cuidado esse arquivo, veremos a primeira vista, vários cabeçalhos de funções, sendo todos eles declarados utilizando o modificador `extern`[11]. Isso é proposital, pois tal modificador indica que uma determinada função estará definida em outro lugar, sendo aquela declaração apenas uma referência para outra declaração em outro arquivo-objeto, que será contemplada durante a *linkagem* no processo de compilação.

Assim, quando seu programa escrito em C for utilizar alguma função da LibClamAv, ele terá que ter alguma referência para essa função, sem precisar ter que implementá-la, uma vez que a implementação se encontra nos arquivos que compõe a LibClamAv.

Além dos cabeçalhos de funções, encontramos alguns parâmetros definidos com a guia `#define` e diversas estruturas de dados. Antes de descrever os cabeçalhos de funções, vamos nos ater nesses parâmetros e nessas estruturas, uma vez que o entendimento desses é primordial para o entendimento correto das funções da API.

Os parâmetros definidos são divididos em três grupos: os códigos de retorno, as opções do banco de dados e as opções do *scan*.

Os códigos de retorno são os valores padronizados de retorno das funções de escaneamento. Deles podemos destacar três principais parâmetros definidos pelas três primeiras linhas transcritas abaixo:

```
#define CL_CLEAN  0
#define CL_VIRUS  1
#define CL_BREAK  2
```

O primeiro, `CL_CLEAN`, é o valor retornado quando o arquivo escaneado encontra-se obviamente “limpo”, i.e., sem nenhum *malware*.

O segundo, `CL_VIRUS`, é o valor retornado quando o arquivo escaneado está infectado por algum tipo de *malware*.

O terceiro, `CL_BREAK`, é o valor retornado quando a função de escaneamento encontrou algum erro, provavelmente pelo fato de o arquivo passado como parâmetro não existir ou não poder ser aberto.

As opções de banco de dados são os valores padronizados que são passados como opção de carregamento da base de dados. Cada valor diferente passado,reflete num tipo de carregamento da base de dados diferente. Não se atendo muito a esse tipo de detalhe, deve-se ressaltar que o mais importante dos parâmetros definidos é o recomendado por padrão, definido como `CL_DB_OPT` .

[11] Para mais detalhes, consulte a página 30 do livro **C Completo e Total** do autor Herbert Schildt.

```
#define CL_DB_STDOPT        (CL_DB_PHISHING                    |
CL_DB_PHISHING_URLS)
```

Por último como grupo de parâmetros definidos, temos as opções de escaneamento. Esses são os valores passados para as funções de escaneamento, definindo o modo de escanemento. Algumas opções desativam a descompressão, outras opções evitam o escaneamento de arquivos PE (não sendo muito útil no Windows) e outras desativam algoritmos especiais de escaneamento para detecção de *worms* complexos.

Por padrão, é utilizada a opção de escanemanento `CL_SCAN_STDOPT`, definida como:

```
#define CL_SCAN_STDOPT       (CL_SCAN_ARCHIVE  |  CL_SCAN_MAIL  |
  CL_SCAN_OLE2 | CL_SCAN_HTML | CL_SCAN_PE | CL_SCAN_ALGORITHMIC
  | CL_SCAN_ELF | CL_SCAN_PHISHING_DOMAINLIST)
```

Explicado para que serve esses parâmetros que são definidos, vamos agora explicar as estruturas de dados. Das quatro estruturas definidas, três são importantíssimas: `cl_cvd`, `cl_limits` e `cl_engine`.

A primeira, `cl_cvd`, representa a estrutura de dados correspondente aos nós armazenados nos arquivos CVDs que compõe a base de dados. Essa estrutura se analisada mais de perto armazena informações como o nome do *malware*, o seu tipo, sua assinatura digital em hexadecimal e uma soma MD5 para validação dessa assinatura.

Essa estrutura não é usada diretamente no levantamento da base de dados e nem no escaneamento propriamente dito, sendo apenas utilizada por programas que operem diretamente com arquivos CVDs como o sigtool mostrado em seções anteriores.

A segunda estrutura é o `cl_limits` que representa algumas informações para limitar o escaneamento. Na seção referente ao clamscan, mostramos um comando para limitar o escaneamento para arquivos de até 100 *megabytes*. Isso é feito com o auxílio dessa estrutura.

A terceira estrutura, a última e mais importante, é a `cl_engine`. Ela representa a *engine* propriamente dita do ClamAv, armazenando parâmetros que serão alterados durante o *scan* e parâmetros para acesso a base de dados. Ao contrário da estrutura `cl_limits`, esta não é alterada diretamente pelo usuário.

Explicados os parâmetros definidos e as estruturas de dados mais importantes, agora podemos descrever melhor as funções que compõe a API da LibClamAv. Como são muitas as funções, descrevermos apenas as mais importantes. Essas podem ser divididas em quatro

grupos: o de controle do CVD, o de controle da base de dados, o de controle da *engine* e o de escaneamento.

As funções de controle do CVD servem obviamente para manipulação dos arquivos CVD permitindo extrair determinadas assinaturas ou adicioná-las a esses arquivos. São essas funções que são usadas na ferramenta sigtool, conforme dito anteriormente. Nesta monografia, essas funções serão apenas mencionadas, pelo fato de não terem sido utilizadas durante a construção do protótipo.

As funções de controle da base de dados são funções relativas a base de dados propriamente dita. São duas, `cl_retdbdir` e `cl_load`, cujos cabeçalhos seguem abaixo.

```
extern const char *cl_retdbdir(void);
extern int cl_load(const char *path, struct cl_engine **engine, unsigned int *signo, unsigned int options);
```

A primeira função, que não possui parâmetros, retorna um ponteiro para uma string contendo o diretório padrão da base de dados, definido durante a instalação da LibClamAv.

A segunda função é a responsável pelo levantamento da base de dados. É a primeira função a ser utilizada durante o procedimento de escaneamento, sendo usada logo depois de se declarar uma estrutura cl_engine que será passada como parâmetro para armazena a *engine*.

A tabela abaixo mostra os diversos parâmetros dessa função e seus respectivos significados:

Parâmetro	Significado
`path`	Ponteiro para uma string contendo o caminho completo da localização da base de dados.
`engine`	Ponteiro para uma estrutura de dados que armazenará a *engine*.
`signo`	Ponteiro para um inteiro que armazenará o total de assinaturas que serão carregadas com esta função.
`options`	Opções para o carregamento do banco.

TAB. 3-2: Significado dos parâmetros passados para a função cl_load da LibClamAv

O retorno dessa função é um número inteiro, sendo zero quando se há sucesso no levantamento da base de dados.

As funções de controle da *engine* são as relativas a inicialização e liberação da *engine*. Duas dessas funções que podemos destacar são: `cl_build` e `cl_free`.

```
extern int cl_build(struct cl_engine *engine);
```

```
extern void cl_free(struct cl_engine *engine);
```

A função `cl_build` é a responsável pela inicialização da *engine*. Esse é o segundo passo que deve ser realizado para se fazer o escanemanento, sendo precedida apenas pelo levantamento a base de dados.

Ela possui como parâmetro um ponteiro para a estrutura de dados contendo a *engine* (a mesma estrutura de dados passada como parâmetro na função `cl_load`). O retorno dessa função é um valor inteiro que, assim como na função `cl_load`, zero representa sucesso.

A função `cl_free` é a responsável pela liberação de memória da *engine*. Isso é um procedimento importantíssimo visto que a base de dados do ClamAv possui vários *megabytes* de memória e por isso é muito importante que esse espaço de memória seja devolvido ao sistema operacional quando não se precisa mais dele.

Esta função recebe como parâmetro também um ponteiro para a estrutura de dados contendo a *engine* a ser liberada. Ela não possui retorno.

Por último, temos o grupo mais importante da API da LibClamAv: o grupo das funções de escaneamento do banco de dados, cuja finalidade se propõe esta biblioteca. Ele é constiuído das funções `cl_scandesc` e `cl_scanfile`, cujos cabeçalhos seguem abaixo:

```
extern int cl_scandesc(int desc, const char **virname, unsigned long int *scanned, const struct cl_engine *engine, const struct cl_limits *limits, unsigned int options);
extern int cl_scanfile(const char *filename, const char **virname, unsigned long int *scanned, const struct cl_engine *engine, const struct cl_limits *limits, unsigned int options);
```

A diferença entre as duas funções é que a `cl_scanfile` recebe como parâmetro um ponteiro para uma string contendo o nome do arquivo e `cl_scandesc` um inteiro com um descritor do arquivo já aberto anteriormente.

Esse descritor de arquivo é um inteiro que identifica unicamente o arquivo no sistema operacional. No entanto, vale destacar que esse número só o identifica no Linux. Portanto, a função `cl_scandesc` é inútil no Windows.

A tabela a seguir descreve o significado dos parâmetros dessas duas funções:

Parâmetro	Significado
`desc`	Descritor do arquivo a ser escaneado.
`filename`	Ponteiro para uma string contendo o caminho completo do arquivo a ser escaneado.
`virname`	Endereço do ponteiro que armazenará o endereço da string que contém o nome do vírus (em caso de infecção)
`scanned`	Ponteiro para um inteiro longo que armazenará o tamanho de memória escaneada em *bytes*.
`engine`	Ponteiro para a estrutura de dados contendo a *enginei* que fora inicializada antes.
`limits`	Ponteiro para a estrutura de dados contendo os limites estabelecidos para o escaneamento.
`options`	Opções de escaneamento.

TAB. 3-3: Significado dos parâmetros das funções cl_scandesc e cl_scanfile da LibClamAv

O retorno da função é um inteiro que pode ser: `CL_CLEAN`, em caso de o arquivo não estar infectado; `CL_VIRUS`, em caso de infecção viral; e `CL_BREAK`, em caso de erro.

3.6. PORTABILIDADE DO CLAMAV PARA O WINDOWS

Até agora discutimos o manuseio dos diversos componentes do ClamAv no seu ambiente original, o Linux, mas em momento nenhum comentamos sobre seu funcionamento na plataforma Windows.

Como foi projetado para ser usado no Linux e não no Windows, existem uma gama muito grande de adversidades que são encontradas quando se deseja portá-lo para o Windows.

Na seção anterior, vimos, por exemplo, que a função `cl_scandesc` é inútil no Windows a não ser que seja encontrada alguma forma de emular os descritores de arquivos do Unix/Linux no Windows. Esse é apenas um exemplo de empecilho que encontramos nesse cenário.

Esta seção atual descreverá então os principais projetos para se portar o ClamAv para o Windows, descrevendo qual delas foi usada neste projeto de protótipo de anti-vírus.

Cronologicamente, o primeiro projeto para se portar o ClamAv para o Windows foi a realizada pelo *Submit Open Source Development Group* (SOSDG)[12]. Nesse projeto, a

[12] Para mais detalhes, acesse: http://www.sosdg.org/clamav-win32/

compatibilidade do ClamAv no Windows foi realizada através da emulação do Linux no Windows com o Cygwin.

Esse projeto de compatibilidade teve sucesso. No entanto, a emulação do ambiente Linux no Windows torna o processo de escaneamento do ClamAv mais lento do que a utilização de bibliotecas nativas, que foram usadas nos projetos subseqüentes.

O segundo projeto foi realizado por um indivíduo chamado Boguslaw Brandys da Bransoft[13]. Essa tentativa se propunha a criação de DLLs próprias do Windows que seriam chamadas diretamente pela aplicação a ser portada. Devido a problemas com a GPL e com os desenvolvedores do ClamAv, esse projeto foi interrompido por um bom tempo e atualmente os novos *releases* são muito esparsos, demorando muito para aparecer. Esse projeto teve algum êxito, uma vez que o ClamMail, um programa que integra o protocolo POP3 (de e-mails) com o ClamAv que verifica a existência de *malwares* em e-mails.

O terceiro projeto foi realizado por um italiano chamado Gianluigi Tiesi[14], que compilou a LibClamAv em formato de DLL no Visual Studio. Para realizar a compatibilização, ele se valeu de um outro projeto de portabilidade as chamadas *Posix Threads* (PThreads) para Windows[15].

Essa tentativa obteve um êxito parcial, uma vez que não foi possível portar os componentes relativos ao escaneamento *on-access*, como o clamd e o clamuko. No entanto, tal tentativa obteve bastante êxito na portação dos outros componentes do ClamAv como o clamscan, freshclam e sigtool, sendo, por isso, o projeto de portabilidade adotado pelo ClamWin.

O último projeto foi realizado por Nigel Horne da *NJH Software*. Trata-se de uma melhoria do projeto de Gianluigi Tiesi e da criação de uma interface gráfica para o ClamAv no Windows. Apesar de ser uma melhoria, esse projeto não consegui a portabilidade dos componentes do escaneamento *on-access*. Tal projeto foi o adotado pelo ClamAv como portabilidade oficial do mesmo para o Windows e, assim, batizado de *ClamAv For Windows*[16].

Esses foram os quatro principais projetos para se portar o ClamAv para o Windows. O projeto adotado na construção do protótipo de anti-vírus foi o terceiro, desenvolvido por Gianluigi Tiesi.

---

[13] Para mais detalhes, acesse: http://www.bransoft.com/clamav.html
[14] Para mais detalhes, acesse: http://oss.netfarm.it/clamav/
[15] Para mais detalhes, acesse: http://www.sourceware.org/pthreads-win32/
[16] Para mais detalhes, acesse: http://w32.clamav.net/

O motivo para escolha foi o fato desse projeto também ser adotado pelo ClamWin, que é um projeto de *front-end* gráfico do ClamAv para o Windows. Esse projeto, apesar de não ter sido utilizado na construção do protótipo, teve bastante influência no desenvolvimento deste projeto final de curso. Por isso, na próxima seção será feita uma breve explanação acerca deste projeto.

3.7. CLAMWIN

O ClamWin Free Antivirus[17], como é seu nome completo, é um anti-vírus livre de código aberto desenvolvido para ser usado no Windows e licenciado pela GPL (*GNU General Public License*), sendo baseado no ClamAv.

FIG. 3-2: Logotipo do ClamWin

Segundo seu site oficial, ele apresenta as seguintes funcionalidades:

- Alta capacidade de detecção de vírus e *spywares*;
- Agendamento de escaneamento;
- Atualização automática da base de dados;
- Escaneamento *on-demand*;
- Integração do *scanner* com o menu de contexto do Windows Explorer;
- *Add-in* para integrar o ClamWin com o Ms Outlook;
- *Tray Icon* disponível para acesso das outras funcionalidades.

[17] Para saber mais acesse seu site oficial: http://www.clamwin.com/

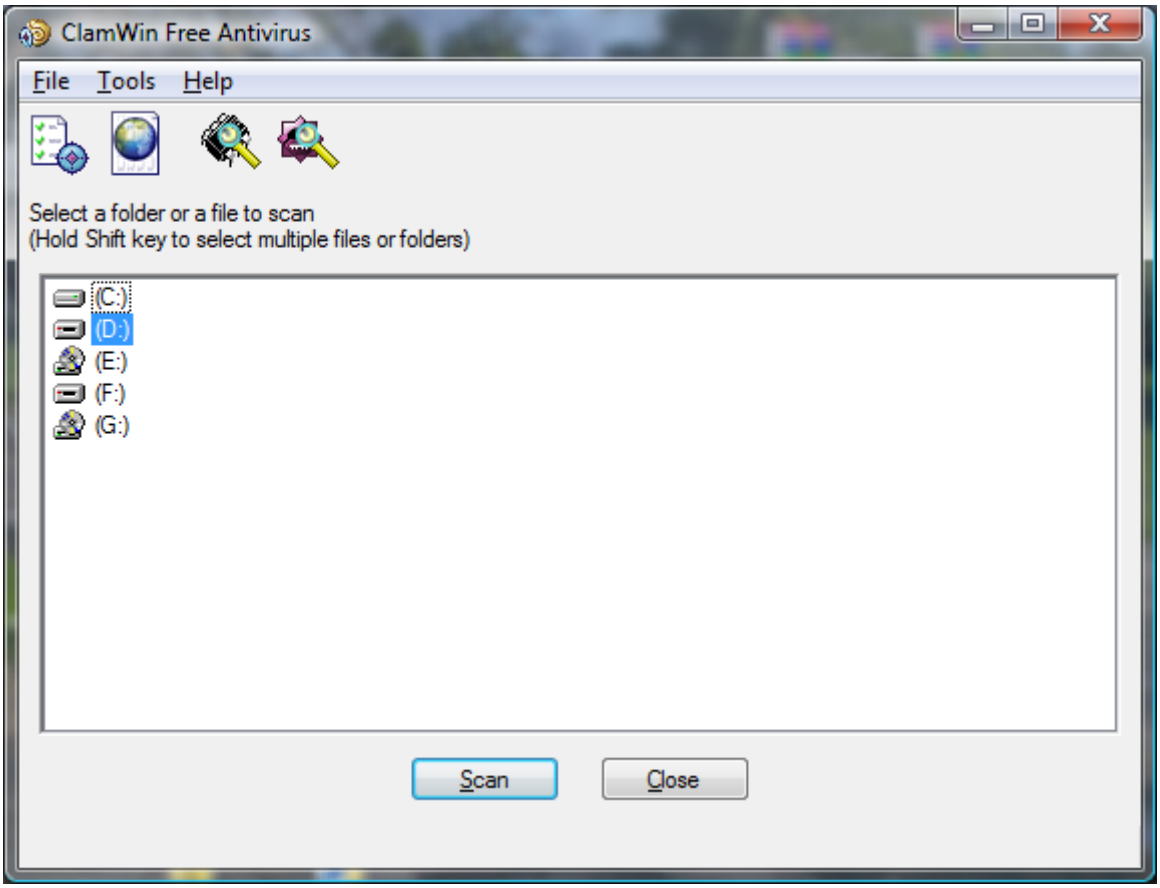

FIG. 3-3: Tela inicial do ClaWin

Basicamente o ClamWin é um *front-end* gráfico das funcionalidades do ClamAv que foram portadas por Gianluigi Tiesi para o Windows. Pode-se notar que as funcionalidades de escaneamento *on-access*, que são desempenhadas pelos componentes clamd e clamuko, não estão presentes ainda no ClamWin.

Analisando mais detalhadamente o código fonte do ClamWin podemos notar que ele é constituído de uma interface gráfica, que foi desenvolvida em C#(.NET); uma camada intermediária, que faz interface com o clamscan e o freshclam, escrita em C++; um módulo denominado *ClamTray*, que se responsabiliza pelo *Tray Icon*[18] do ClamWin; um módulo responsável pela integração com o Windows Explorer; e, por último, de um instalador e *update* do ClamWin, ambos escritos na linguagem NSIS, um linguagem de script de construção de instaladores, que permite criação de diretórios, bem como alteração do registro.

[18] *Tray Icon* é como é chamado os ícones de programas que se situam do lado do relógio do Windows

3.8. PONTOS FORTES E PONTOS FRACOS DO CLAMAV

Até agora foi explicado todo o funcionamento do ClamAv, como é dividido em termos de componentes, como está seu grau de portabilidade para o Windows, mas em nenhum momento foi discutido sua eficiência comparando-o com outros anti-vírus.

É verdade que foi dito que o ClamAv foi considerado como tendo a melhor base de dados de assinaturas digitais virais do mundo, devido a sua atualização, mas não foi abordado nenhum outro ponto do mesmo. Esse é um ponto fortíssimo do ClamAv, mas é óbvio que ele posui outros pontos fracos também.

Nessa seção, abordaremos as vantagens e desvantagens da adoção do ClamAv como anti-vírus, expondo seus pontos fortes e pontos fracos. Os pontos que aqui serão relatados são os resultados uma série de testes com o ClamAv explicitados por um artigo do *site Tech-Pro.Net* [19].

Segue abaixo, dessa maneira, uma lista com diversas modalidades de testes e os seus respectivos resultados.

- **Detecção de *worms***: Foi constatado que o ClamAv é excelente na detecção de *worms*, principalmente os que infectam e-mails (*phishing e-mails*, *e-mail exploitsI*, etc).

- **Detecção de vírus**: Foi constatado que o ClamAv se encontra na média em termos de detecção de vírus, sendo um ponto positivo o fato dele detectar muitos vírus que são difíceis de se encontrar normalmente.

- **Vírus no setor de *boot***: Foi constatado que o ClamAv é **inútil** para se detectar esses tipos de vírus que se encontram na MBR.

- **Falsos Positivos**: Foi constatado que o ClamAv produz ocasionalmente falsos positivos. No entanto, a freqüência desses falsos positivos se encontra dentro de um valor esperado, já que muitos outros anti-vírus produzem muito mais falsos positivos.

- **Restauração de arquivos**: Foi constatado que o ClamAv **não** restaura arquivos infectados, ao contrário de outros anti-vírus. Segundo o artigo do *site*, isso não é de fato um problema, uma vez que raramente esta limpagem do arquivo é eficiente, sendo que muitas vezes ela acaba por corrompê-lo.

- **Velocidade**: Foi constatado que o ClamAv é **lento** em termos de velocidade de *scanning*, se comparado com outros anti-vírus.

[19] http://www.tech-pro.net/clamav.html

- **Uso de recursos do sistema**: Foi comprovado que o ClamAv é **moderado** em termos de uso de memória do sistema e CPU, sendo este último recurso muito gasto quando se escaneia arquivos comprimidos.

- **Atualização da base de dados**: O *site* confirma o que já havia sido dito anteriormente, relatando que as atualizações do ClamAv são rápidas e freqüentes. O artigo também relata que a disposição dos *mirrors* torna o *download* das atualizações muito fáceis de serem executados, mesmo para quem possue conexão com a internet *dial-up*.

Assim, com base nas conclusões aqui expostas, vemos que o ClamAv é uma excelente ferramenta como anti-vírus, mas não devemos esquecer que ele precisa melhorar muito em alguns aspectos, sendo nesses muito inferior a outros anti-vírus.

4. DRIVERS NO WINDOWS

4.1. ARQUITETURA DO WINDOWS NT

O Windows NT (*New Technology*) foi criado pela Microsoft para ser o novo padrão de sistema operacional da empresa no mercado, rompendo com a herança da antiga família de sistemas operacionais descendentes do Windows 95. Ele deu origem também à sua própria família de sistemas operacionais, com descendentes como o Windows 2000, XP e, mais recententmente, o Vista.

Ele possuía uma série de requisitos de projetos, dos quais cabe destacar: operar completamente em 32 bits, ser portável para diversas plataformas de *hardware*, ser confiável e robusto, podendo ser utilizado como servidor ou *workstation*. Esses requisitos foram atendidos com sucesso através da adoção da arquitetura que será a seguir explicada.

Antes de explicarmos a arquitetura do Windows NT, devemos esclarecer os conceitos de modos de execução, especificamente o *kernel mode* e o *user mode*. Esses modos de execução dizem respeito ao grau de privilégio de acesso aos recursos do sistema operacional e do *hardware* subjacente. Essa distinção nesses modos foi criada para proteger os componentes críticos e confiáveis do sistema operacional de falhas (erros, corrupção de memória, operações ilegais, etc) de aplicações de usuário que poderiam, de outra forma, prejudicar o funcionamento do sistema como um todo, causando até mesmo a sua falha completa (*system crash*).

O *kernel mode* é o modo de execução mais privilegiado, que permite acesso e controle total dos recursos da máquina. Todos os componentes confiáveis e críticos do sistema rodam nesse modo.

O *user mode* é o modo de execução menos privilegiado. As aplicações comuns rodam nesse modo e não podem fazer, diretamente, acesso aos serviços e recursos do SO e da máquina.

Para fazer acesso aos serviços e recursos do sistema, existem o mecanismo de *system calls*. Ao ocorrer uma *system call*, o sistema operacional faz uma troca de contexto, do *user mode* para o *kernel mode*, executa o código adequado do sistema operacional, e troca de volta do *kernel mode* para o *user mode*, retornando os resultados adequados e devolvendo o controle para a aplicação.

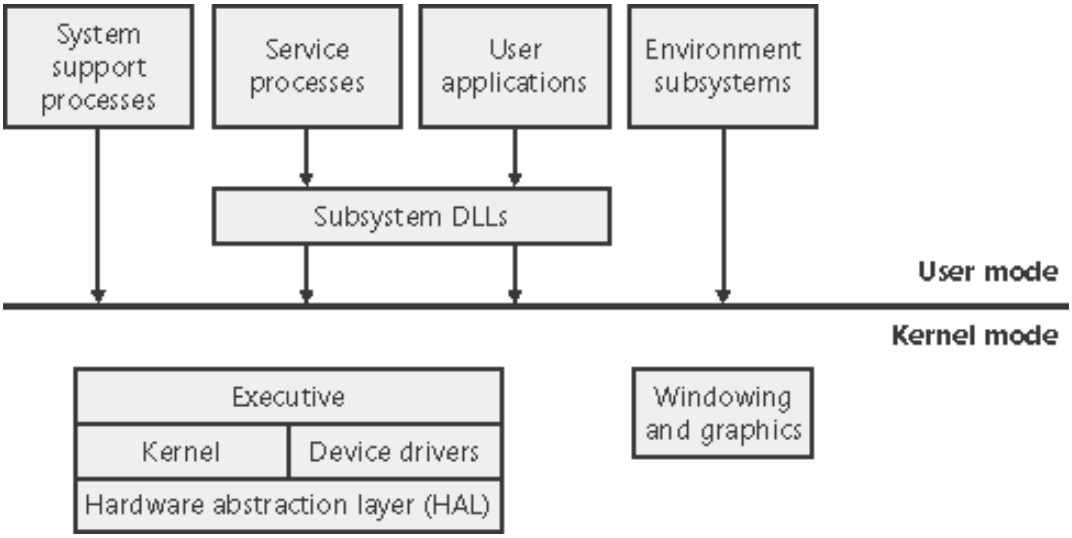

FIG. 4-1: Visão simplificada da arquitetura do Windows NT

Os componentes *user mode* da arquitetura NT são:

- *System support processes*: processos de suporte ao sistema, como o processo de *logon* e o *session manager*, que não são iniciados pelo *service control manager*;
- *Service processes:* processos que constituem serviços do sistema, com o requisito de serem executados de forma independente dos *logons* de usuário;
- *User applications*: aplicações e programas que um usuário usa e/ou interage direta ou indiretamente com para executar suas diversas tarefas no sistema;
- *Environment subsystems*: subsistemas responsáveis por implementar o suporte à interface de diversos sistemas operacionais. Originalmente continha os seguintes subsistemas: Windows, OS/2 e POSIX. A partir do Windows XP, apenas o subsistema do Windows vinha com o produto base.

Os componentes *kernel mode* da arquitetura NT são:

- *Executive*: contém a base dos serviços do sistema operacional, como gerenciamento de memória, gerenciamento de processos e threads, *networking*, I/O, comunicação interprocesso;
- *Kernel*: consiste de funções de baixo nível do sistema operacional, como escalonamento de threads, *dispatching* de interrupções e exceções de *hardware*, sincronização de múltiplos processadores. O *Kernel* também provê rotinas e

objetos básicos que o *Executive* utiliza para implementar construtos de mais alto nível;

- *Device drivers*: *drivers* responsáveis por traduzir os *requests* de I/O do usuário em comandos específicos que podem ser entendidos pelo dispositivo de *hardware* em questão. Além desses *drivers*, também se encontram nessa categoria os *file system drivers* e os *network drivers*;
- *Hardware abstraction layer* (HAL): camada de *software* do sistema que é responsável por isolar os demais componentes de *kernel mode* do hardware subjacente, abstraindo completamente as diferenças e especificidades dele, e oferecendo uma interface de acesso uniforme. É a implementação da HAL para diversas plataformas de *hardware* que permite que o Windows NT seja portável;
- *Windowing and graphics system*: implementa as funções da interface com o usuário (GUI), como tratamento de janelas, controles de interface e *drawing* de componentes gráficos.

Para o presente trabalho cabe ressaltar, novamente, que as aplicações de usuário, conforme vimos na arquitetura acima, não podem fazer acesso aos serviços de *kernel mode* do sistema operacional diretamente. Isso deve ser feito através do uso de interfaces para esses serviços providas em DLLs (*dynamic link libraries*), sendo a principal dessas a Ntdll.dll.

De fato, o *kernel mode* possui uma série de particularidades em relação ao *user mode*. Como exemplo, temos o fato de todos os componentes *kernel mode* do sistema compartilharem o mesmo espaço de memória entre si. Isso implica que **qualquer erro de alocação ou corrupção de memória em *kernel mode* pode provocar um *system crash***. Por essa e por outras razões, temos a premissa de que os componentes que operam em *kernel mode* são componentes confiáveis. Mais adiante nesse trabalho, na seção 4.6 deste documento, teceremos algumas recomendações acerca de programação em *kernel mode*.

Para o presente trabalho, faz-se deveras importante fixar tais conceitos, pois estaremos construindo, para compor nossa solução final do anti-vírus, um *file system driver*, que, como vimos, pode ser considerado um *device driver* e opera em *kernel mode*. **Um *device driver* com erros graves de implementação pode levar facilmente a constantes falhas e prejuízo para o sistema operacional como um todo**.

Para mais informações sobre o Windows NT, seus descendentes, seu funcionamento e sua arquitetura, consultar o excelente livro "*Microsoft Windows Internals*", constante da bibliografia deste trabalho.

4.2. WINDOWS DRIVER MODEL

Windows Driver Model (WDM) é um modelo utilizado pelos *drivers* do Windows, a partir do Windows NT, substituindo o antigo modelo, chamado VxD, que era utilizado nos Windows 3.x e Windows 9.x. O objetivo do WDM é padronizar os requisitos dos diversos tipos de drivers em um modelo unificado, reduzindo a quantidade de código necessário para a criação dos mais diversos tipos de *drivers*.

O WDM foi projetado para que *drivers* criados com ele sejam *forward-compatible*, ou seja, que eles possam rodar em uma versão posterior à versão do Windows na qual eles foram desenvolvidos. Dessa forma, um *driver* WDM para Windows 2000, em teoria, funcionará normalmente para o Windows XP, por exemplo. No entanto, apesar de ser compatível com as versões posteriores de sistemas operacionais, o *driver* não poderá usufruir de qualquer recurso que tenha sido introduzido nessas versões posteriores. De maneira geral, um *driver* WDM não é *backward-compatible*, ou seja, não existe qualquer garantia de que ele funcione em versões anteriores à versão do Windows na qual ele foi projetado.

O WDM está organizado em uma hierarquia de drivers, organizados em pilhas, chamadas *device stacks*, cada uma associada a um dispositivo. Os *drivers* podem ser dos seguintes tipos:

- *Bus drivers*: são *drivers* de mais baixo nível, que operam diretamente em barramentos de *hardware*, como barramentos PCI e USB, e dispositivos conectados a esses barramentos;
- *Function drivers*: são os *drivers* que realmente implementam as funcionalidades, associadas ao dispositivo em questão. Eles são os responsáveis por traduzir as requisições de I/O em comandos que podem ser entendidos pelo *hardware* do dispositivo;
- *Filter drivers*: são *drivers* que se encontram no meio da *device stack* e filtram as requisições que chegam, realizando processamentos adicionais relevantes e podendo modificá-las, respondê-las ou até mesmo abortá-las.

Os *drivers* WDM se comunicam entre si através do envio de *I/O Request Packets* (IRP). Os IRPs não só carregam informações de requisições de I/O que aplicações enviam aos

drivers, elas também carregam requisições de *Plug and Play*, requisições de gerenciamento de energia, requisições de *Windows Management Instrumentation* (WMI), notificações ou mudanças no status de dispositivos e consultas (*queries*) sobre recursos de dispositivos e *drivers*. A figura a seguir mostra como um IRP é tratado em uma *device stack*:

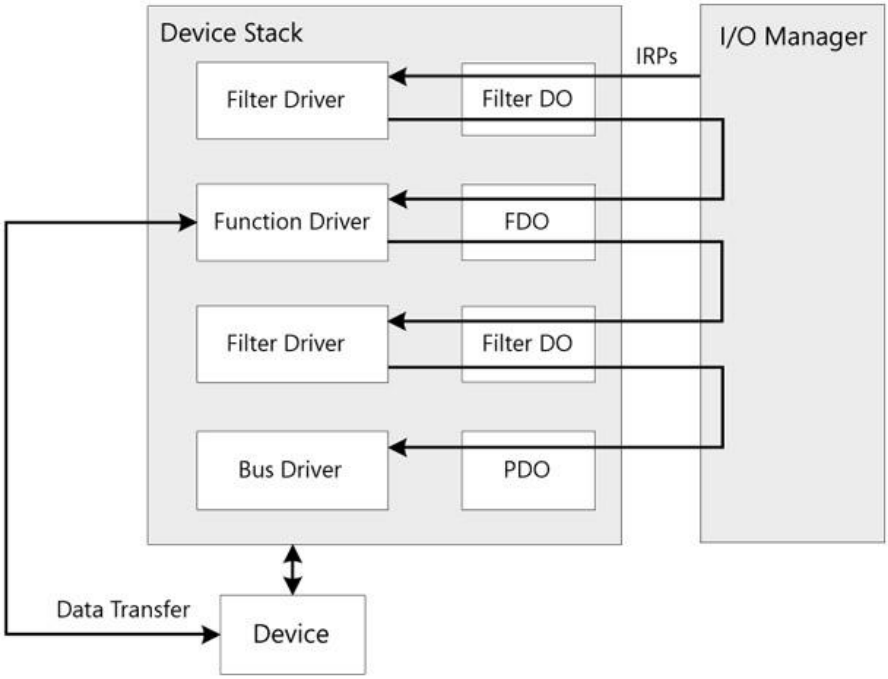

FIG. 4-2: Como um IRP é tratada em uma *device stack*

Se um *driver* mais acima puder satisfazer o IRP, ele responde ao *I/O Manager*, completando a requisição. Com essa ação, o IRP é devolvido ao *I/O Manager*, que repassa qualquer dado requisitado ao cliente. Se, no entanto, um *driver* não puder satisfazer a requisição, ele realiza o processamento necessário sobre a IRP e devolve-a ao *I/O Manager*, que a repassa então ao próximo *driver* mais abaixo na *device stack*. Em algum momento, a IRP chegará a um *driver* que pode satisfazer a requisição. Ele o fará e devolverá o IRP ao *I/O Manager*, passando também quaisquer dados que devam ser retornados ao cliente. *Drivers* podem, opcionalmente, realizar algum tipo de processamento após a requisição ter sido satisfeita. Nesse caso, o *I/O Manager* repassa a tais *drivers* o IRP para pós-processamento, na ordem inversa em que os *drivers* se encontram na *device stack*.

Embora o WDM tenha representado um grande avanço no sentido de facilitar o desenvolvimento de drivers, ele sofreu diversas críticas, das quais cabe destacar:

- É muito complexo e de difícil aprendizado;

- Cancelamento de operações de I/O é muito difícil de ser implementado corretamente;
- Interações com eventos de gerenciamento de energia e de *Plug and Play* são difíceis de serem implementados;
- Não existe suporte para *drivers* que se situam completamente em *user mode*;
- Milhares de linhas de código eram requeridas para construção de cada *driver*, mesmo para os mais simples.

4.3. WINDOWS DRIVER FOUNDATION

Para resolver os problemas da WDM acima descritos, a Microsoft realizou um grande *refactor* na WDM, criando a *Windows Driver Foundation* (WDF), com o objetivo de reduzir a complexidade do processo de criação de *drivers* e permitir a criação de *drivers* mais robustos. A WDF compreende não só um modelo de *drivers*, mas também *frameworks* para desenvolvimento e ferramentas para verificação e teste.

As vantagens da utilização da WDF, segundo a Microsoft, são as seguintes:

- Reduz a quantidade de código necessário para o desenvolvimento de um *driver*;
- Produz *drivers* mais robustos e de melhor qualidade;
- Melhora a segurança e estabilidade do sistema;
- Permite a descoberta de *bugs* em *drivers* através do uso de ferramentas de análise estática de código;
- Reduz as chances de um *device driver* causar um *system crash*;
- Provê melhor gerenciamento de energia;
- Provê uma melhor experiência acerca de *pluggable devices*.

A idéia básica da WDF é permitir que um desenvolvedor crie um *driver* inicialmente simples, porém funcional, e depois, conforme a necessidade, acrescente de forma incremental funcionalidades relevantes, até completar o *driver*. Assim, o desenvolvedor não precisa conhecer todos os conceitos e detalhes técnicos necessários para o desenvolvimento do *driver*: ele pode começar com os conceitos mais básicos e ir aprendendo os conceitos necessários à medida que vai desenvolvendo incrementalmente as funcionalidades do *driver*.

A WDF, ao contrário da WDM, adota uma arquitetura baseada em objetos, eventos e *callbacks*. Os objetos são abstrações da *framework* das entidades com as quais o *driver* interage durante o seu ciclo de vida: objetos de *drivers*, objetos de dispositivos, objetos de

requisição de I/O, etc. Cada objeto desses possui atributos associados e um ciclo de vida, que é também gerenciado pela própria framework, liberando os recursos alocados do sistema quando eles não são mais necessários. Os eventos são ocorrências do sistema que estão relacionadas a um dado objeto e que um *driver* pode optar por tratar ou não. As *callbacks* são as funções que devem ser chamadas quando o evento ocorre, de forma a realizar o seu tratamento. A figura a seguir mostra uma visão conceitual da arquitetura da WDF:

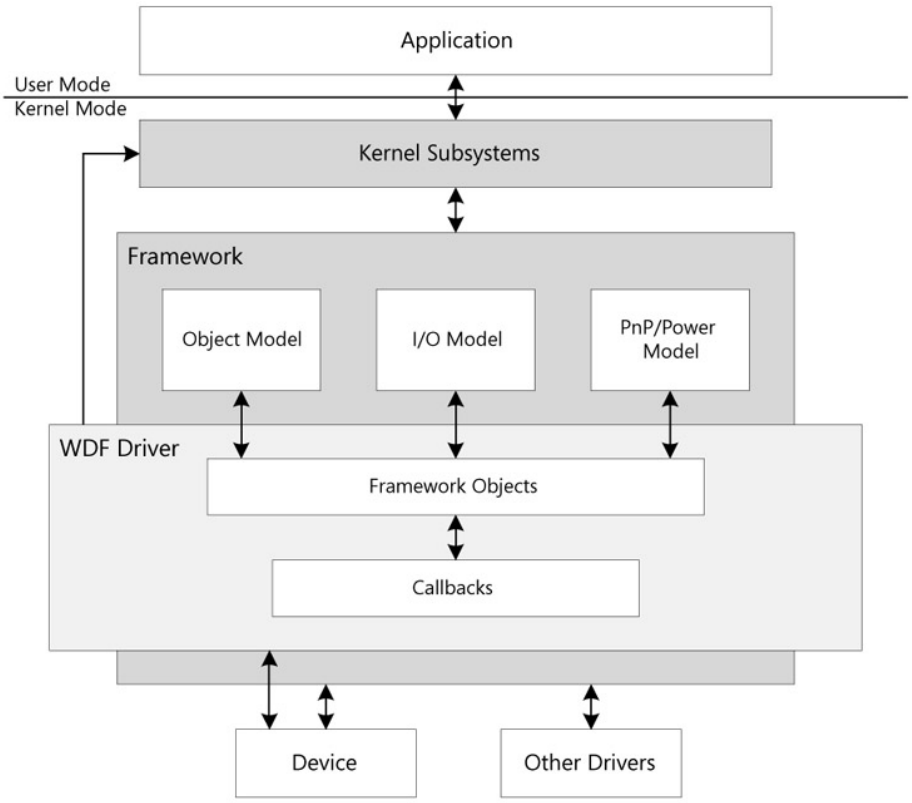

FIG. 4-3: Visão conceitual da arquitetura da WDF

A WDF possui dois *frameworks* para desenvolvimento de *drivers*: *Kernel Mode Driver Framework* (KMDF) e *User Mode Driver Framework* (UMDF).

A KMDF é a *framework* que deve ser utilizada para o desenvolvimento de *drivers* que se situam em *kernel mode*. Ela possui uma correlação direta com o modelo da WDM, provendo de fato uma camada de abstração em cima dele. Os *drivers* desenvolvidos com a KMDF devem ser escritos apenas em C, por questões de desempenho e portabilidade. No entanto, como foi dito anteriormente, a KMDF é toda baseada em objetos (no caso de C, isso implica não em objetos reais, como em C++, mas em estruturas de dados, que tem atributos e um tempo de vida associado). Na realidade, não se utiliza o C padrão ANSI, mas um C estendido

pela Microsoft, que contém extensões úteis, como construtos *try-except*[20] e *try-finally*[21] para controle mais adequado dos erros (exceções) que possam ocorrer no sistema. Cabe ainda lembrar que na programação em *kernel mode*, não é possível fazer uso de muitas das funções da biblioteca padrão do C, porque ela é uma biblioteca construída para uso por aplicações em *user mode* que, em última instância, podem indiretamente vir a chamar os serviços do *Executive* e *Kernel* do Windows, prejudicando o desempenho do sistema e abrindo possibilidades para a ocorrência de erros e falhas.

A UMDF é a *framework* que deve ser utilizada para o desenvolvimento de *drivers* que se situam em *user mode*. A possibilidade de escrever *drivers* cuja maior parte da funcionalidade se encontra em *user mode* é uma novidade da WDF que não existia no WDM. Os *drivers* na UMDF são escritos em C++ (embora possam ser também escritos em C, o que é incomum) e adotam uma interface de *callbacks* baseadas no padrão *Component Object Model* (COM). Além disso, como se trata de um componente *user mode*, o *driver* pode fazer uso de funções da biblioteca padrão e da API do Windows (Win32 API). No entanto, ele não possui acesso a determinados recursos, que são recursos disponíveis apenas em *kernel mode*, como, por exemplo, interrupções, *Direct Memory Access* (DMA), acesso a registradores, etc. Ao contrário do que ocorre em *drivers* KMDF, um erro em um *driver* UMDF não causa um *system crash*, podendo o sistema inclusive tentar a recuperação do *driver* para um estado válido.

A WDF é disponibilizada para instalação sob a forma do *Windows Driver Kit* (WDK), um pacote contendo todas as ferramentas, bibliotecas e documentação que são necessárias para o desenvolvimento, teste e depuração de *drivers*, além de conter também muitos exemplos de *drivers*. O processo de instalação do WDK está descrito em um apêndice, na seção 11.1 deste documento.

O WDK apresenta também o conceito de *building environment*. Um *building environment* é um ambiente de *build* preparado para construir os binários necessários de um *driver* voltados para uma determinada plataforma específica.

A WDF foi feita de tal forma, que a partir de um mesmo código-fonte, é possível, em teoria, utilizando tais *build environments*, gerar binários funcionais para praticamente todas as versões atuais do Windows (2000, XP, Server 2003, Vista e Server 2008), voltadas para as

[20]Para mais informações sobre o construto *try-except* em C, acesse http://msdn.microsoft.com/en-us/library/zazxh1a9.aspx.
[21]Para mais informações sobre o construto *try-finally* em C, acesse http://msdn.microsoft.com/en-us/library/yb3kz605.aspx.

principais arquiteturas arquiteturas do mercado (x86, x64 e IA-64). À medida que o WDK vai sendo atualizado e as versões dos sistemas Windows vão evoluindo, os *build environments* mais antigos vão sendo descontinuados.

Para mais informações sobre o WDK, consulte a sua documentação, disponível na instalação ou em versão *online* dentro da MSDN. Consulte também o excelente livro "*Developing Drivers with the Windows Driver Foundation*", constante da bibliografia deste trabalho.

4.4. FILE SYSTEM FILTER DRIVERS

Um *file system filter driver*, como o próprio nome indica, é um *filter driver* (e, portanto, um *kernel mode driver*) que filtra as requisições destinadas a algum *file system*, materializado na forma de um *drive* de um dispositivo de memória secundária.

Através de um *file system filter driver*, o sistema pode adicionar, de forma completamente transparente ao usuário, uma série de funcionalidades úteis a *file systems*. Como exemplo, podemos citar a criptografia e compressão nativa de sistemas de arquivo NTFS.

Um *file system filter driver* provê a solução perfeita para a interceptação das requisições de acesso ao *file system*, necessária para prover a *feature* de *on-access scanning* que desejamos implementar para o nosso anti-vírus estático em ambiente Windows.

Para construir um *file system filter driver*, existe um modelo e API específicos, largamente baseados nos providos pela WDM.

O modelo funciona de forma com o uso de callbacks, que especificam o processamento realizado pelo *file system filter driver* quando determinados eventos do *file system* ocorrem. O modelo provê também a capacidade de especificar em que ordem os *file system filter drivers* serão carregados no sistema.

Embora tal modelo tenha tido sucesso em prover as funcionalidades necessárias para a criação dos mais diversos tipos de *file system filter drivers*, tal como o WDM, ele mostrou-se muito complexo e pouco flexível.

Seguindo a mesma filosofia adotada no *design* da WDF, a Microsoft efetuou então um grande *refactor* nesse modelo, abstraindo as partes comuns a diversos *file system filter drivers*, principalmente as que dizem respeito ao gerenciamento do tempo de vida do *driver* e interação com outros objetos, e deixando ao desenvolvedor apenas a tarefa de especificar o código referente às *callbacks* e outras partes específicas de cada *driver*. O novo modelo criado nesse processo foi batizado de *FS Minifilter Driver Model*, sendo os *drivers* criados com ele

batizados de *minifilters*, nome inspirado no fato do código desses *filter drivers* ser muito menor que o código de seus equivalentes no antigo modelo.

O *FS Minifilter Driver Model* é hoje o padrão para desenvolvimento de *file system filter drivers*, sendo distribuída juntamente com a WDF, sob a forma de WDK[22], e tendo, naturalmente, grande interação com ela e com os demais componentes *kernel mode* do Windows. O modelo antigo, agora batizado *Legacy Filter Driver Model*, continua, no entanto, sendo suportado por razões de compatibilidade com os *filter drivers* anteriormente criados, mas será descontinuado no futuro.

4.5. MINIFILTERS

A Microsoft, ao propor o novo modelo de criação de *file system filter drivers*, o chamado *FS Minifilter Driver Model*, apresentou as seguintes vantagens da utilização desse modelo ao invés do *Legacy Filter Driver Model*:

- Melhor controle sobre a ordem de carregamento dos *filter drivers*;
- Habilidade de descarregar um *filter driver* com o sistema rodando;
- Habilidade de processar apenas as requisições de I/O que interessam ao *driver*;
- Uso mais eficiente da memória da *Kernel Stack*;
- Menos código redundante;
- Complexidade reduzida;
- Facilidade de adicionar, de forma incremental, suporte a outras requisições no futuro;
- Melhor suporte a múltiplas plataformas;
- Melhor suporte para comunicação com aplicações que operam em *user mode*.

Esse será, portanto, o modelo que estaremos utilizando nesse trabalho para implementação do componente responsável por filtrar as requisições dirigidas ao *file system* do sistema. A seguir, explicaremos mais em detalhes a arquitetura geral adotada por esse modelo.

A arquitetura do *FS Minifilter Driver Model* baseia-se em um componente especial, chamado *Filter Manager*. O *Filter Manager* é um *driver kernel mode* que atende ao *Legacy Filter Driver Model*. Esse componente agrega todas as funcionalidades comumente requeridas

[22] Mais especificamente falando, a parte do WDK que contém o material necessário para o desenvolvimento de *minifilters* é chamada de *Installable File Systems Kit* (IFSK).

para o funcionamento dos *legacy filters*, deixando ao desenvolvedor apenas a tarefa de incluir o tratamento das funcionalidades particulares do seu *minifilter*. O *Filter Manager* é responsável também por manter a lista dos *minifilters* atualmente ativos no sistema, carregá-los na ordem adequada, gerenciar seu ciclo de vida e repassar, na devida ordem, as requisições que chegam ao *file system* aos *minifilters* devidamente registrados junto a ele.

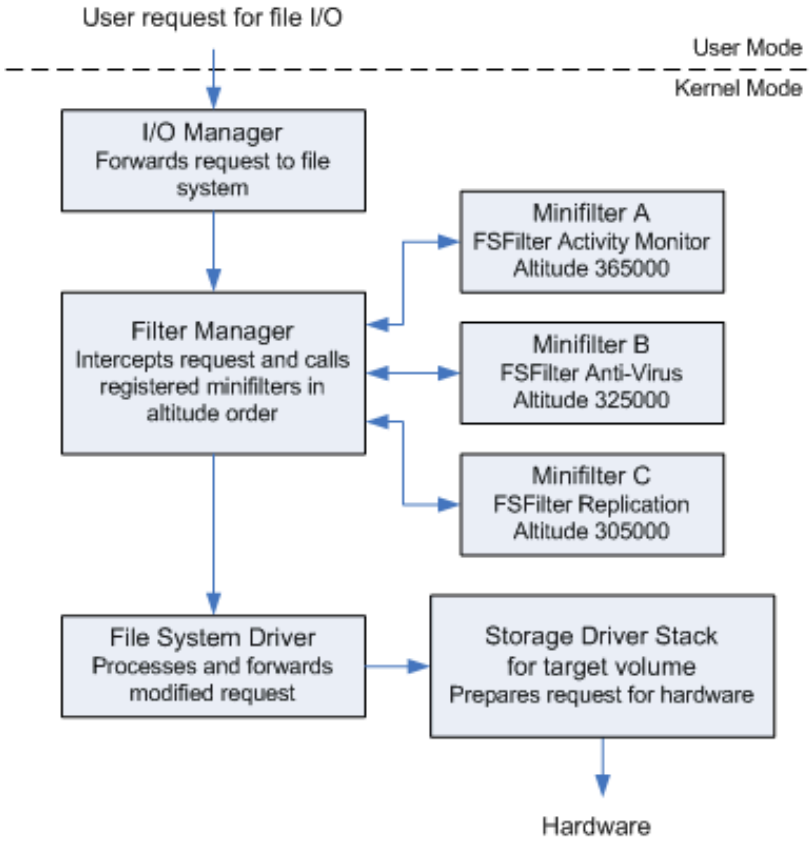

FIG. 4-4: Esquema simplificado do posicionamento do *Filter Manager* e dos *minifilters* na pilha de I/O

O *Filter Manager* vem instalado com o Windows, mas só se torna ativo no sistema no momento em que o primeiro *minifilter* é carregado. Ele então se acopla à pilha do *file system* para um determinado *volume* (i.e., um *drive*) do sistema. Um *minifilter* pode então se acoplar a esse *volume* indiretamente, através do seu registro junto ao *filter manager*, que então repassará a ele as requisições de I/O endereçadas àquele *volume*.

A ordem em que os *minifilters* se acoplam ao *volume* é determinada por um identificador, numérico de ponto-flutuante com precisão infinita, chamado *altitude*. O acoplamento de de um dado *minifilter driver* a um dado *volume* em uma *altitude* particular é chamado de instância (ou *instance*) do *minifilter*.

A *altitude* garante que uma *instance* de um *minifilter* é sempre carregada na ordem apropriada relativamente a instances de outros *minifilters*. Quanto menor o valor da altitude, mas cedo é carregada a *instance* e mais perto do *file system* ela fica. Se um *minifilter* for descarregado (*unloaded*) e depois recarregado (*reloaded*), ele será carregado na mesma *altitude* que estava antes de ser descarregado.

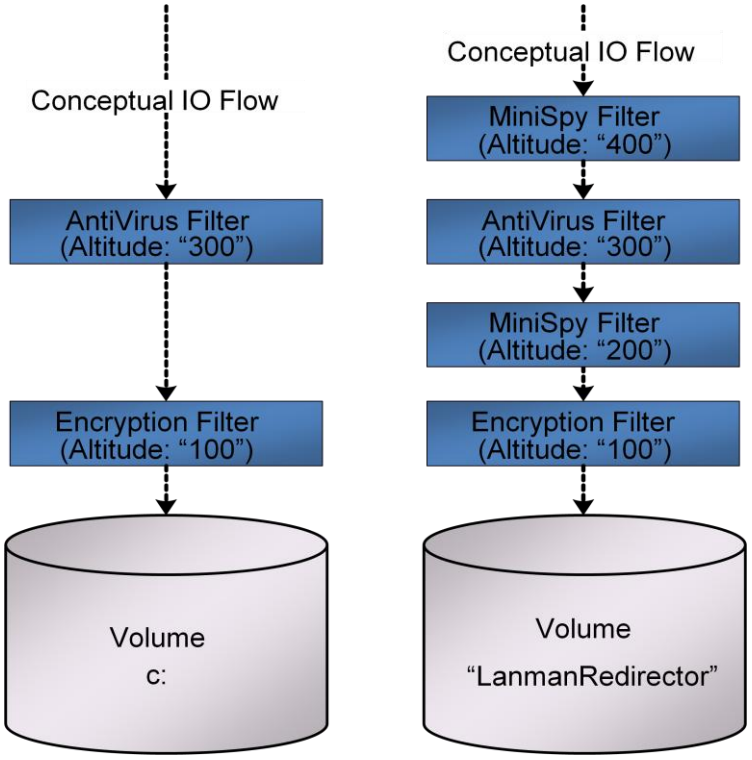

FIG. 4-5: Exemplo de como a *altitude* influencia a ordem relativa em que os *minifilters* são dispostos na pilha de I/O para o dado *volume*

Para uniformizar os valores e evitar conflitos e problemas desenecessários entre devido a escolhas arbitrárias de *altitudes*, elas são atribuídas e gerenciadas pela própria Microsoft[23]. Todo *minifilter* que vá ser comercializado ou distribuído ao público deve possui uma *altitude* válida, devidamente atribuída pela Microsoft. A tabela a seguir mostra exemplos de valores possíveis para as *altitudes* e os grupos de *minifilters* que os usam:

[23] O requerimento de um valor de *altitude* para o *minifilter* pode ser feito através do formulário em http://connect.microsoft.com/content/content.aspx?ContentID=2512&SiteID=221.

Nome do grupo	Intervalo de *altitude*	Descrição do grupo
FSFilter Top	400000-409999	*Minifilters* que devem ser carregados acima de todos os outros.
FSFilter Anti-Virus	320000-329999	*Minifilters* que implementam funcionalidades relacionados a detecção e desinfecção de vírus em operações de I/O em arquivos.
FSFilter Content Screener	260000-269999	*Minifilters* que impedem a criação de determinados tipos de arquivos ou que contenham determinado conteúdo.
FSFilter Compression	160000-169999	*Minifilters* que realizam compressão e descompressão de dados de uma operação de I/O em arquivos.
FSFilter Encryption	140000-149999	*Minifilters* que encriptam e decriptam os dados durante uma operação de I/O em arquivos.
FSFilter Bottom	40000-49999	*Minifilters* que devem ser carregados abaixo de todos os outros.

TAB. 4-1: Grupos de *minifilters* e seus respectivos intervalos de valores para *altitude*

Os *minifilters* podem ser utilizados juntamente com *legacy filters*. Para tal, o *Filter Manager* forma *frames*, que são grupos de *minifilters* que se acoplam à pilha de I/O de um dado *volume* em uma determinada posição, atuando como um *legacy filter*. Os *frames* podem ser intercalados por *legacy filters* na pilha, uma vez que os *frames* também são interpretados como *legacy filters* tradicionais.

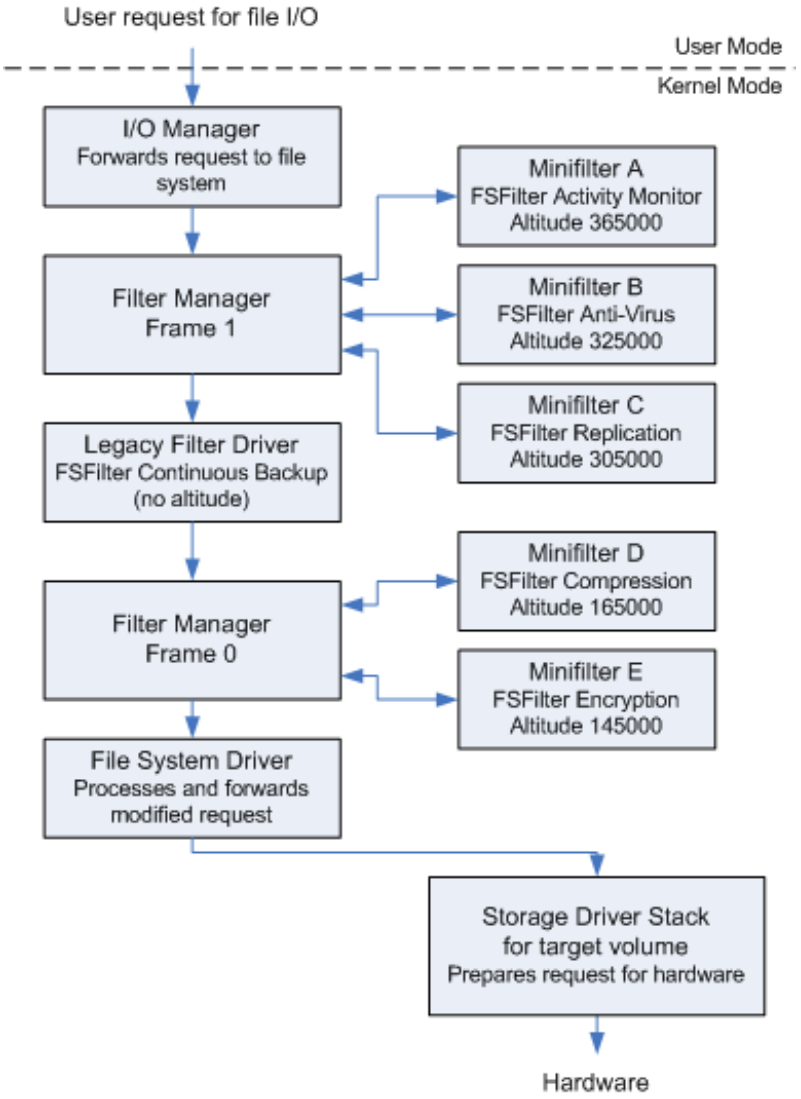

FIG. 4-6: Pilha de I/O com dois *frames*, contendo vários *minifilters*, e um *legacy filter*

Minifilters são instalados via arquivos INF. O arquivo INF inclui todas as informações do *minifilter* necessárias à instalação e operação do *Filter Manager*, tais como: nome, versão e *provider* do *driver*; nome e localização dos arquivos que o compõem e devem ser instalados; se ele deve ser carregado sob demanda ou automaticamente com o início do sistema; a sua *altitude*; as chaves do registro que devem ser adicionadas; as *instances* do *minifilter* e suas configurações. Além disso, o arquivo INF contém informações que podem ser utilizadas para desinstalar o *minifilter*[24].

Um *minifilter*, quando não for carregado automaticamente, pode ser carregado via uma requisição de *start* de serviço (`net start <filter_name>` ou `sc start`

[24] Para mais informações sobre arquivos INF de *minifilters*, acessar http://msdn.microsoft.com/en-us/library/ms793586.aspx.

<filter_name>) ou programaticamente, através de chamadas à API de *minifilters* (funções `FltLoadFilter()` e `FilterLoad()`[25]).

O arquivo fonte do *minifilter* deve efetuar diretivas `#include`, de forma a incluir os arquivos de *header* `fltKernel.h` e `ntifs.h`, que contêm todas as definições das estruturas e tipos necessários para a criação do componente *driver*. Um componente *user mode* que deseje fazer uso da API dos *minifilters* deve incluir o arquivo de *header* `fltuser.h`.

A seguir, explicaremos mais em detalhes a implementação de um *minifilter*. Cabe destacar que não é a intenção deste trabalho oferecer um tratado completo, esgotando tal extenso assunto, ou ainda substituir a devida documentação da WDK, mas sim lançar um pouco de luz sobre esse tópico, dando ao leitor conhecimento suficiente para começar a construção de um *minifilter*.

Todo *minifilter* deve especificar uma função chamada `DriverEntry`. A função `DriverEntry` é chamada sempre que o *minifilter* é carregado. Importante lembrar que isso só ocorre uma única vez, independente do número de *instances* que o *minifilter* possa ter carregadas no sistema. Nessa função, pode-se especificar qualquer inicialização que seja necessária e que se aplique a todas as *instances*.

O *minifilter* deve chamar a função `FltRegisterFilter` dentro de `DriverEntry`. A função `FltRegisterFilter` é utilizada para registrar o *minifilter* junto ao *Filter Manager*. Ela recebe como um de seus parâmetros uma estrutura `FLT_REGISTRATION`, que contém a rotina que deverá ser chamada para descarregar (*unload*) o *minifilter*, funções de configuração e destruição de cada *instance* (geralmente chamadas de `InstanceSetup` e `InstanceQueryTeardown`, respectivamente) e um array de estruturas `FLT_OPERATION_REGISTRATION`, que especificam, cada uma, uma determinada requisição de I/O na qual o *minifilter* está interessado, *callbacks* para tratá-la e *flags*, indicando em que situações a *instance* deve ser notificada.

A função `InstanceSetup`, que deve possuir assinatura compatível com a definida pelo tipo `PFLT_INSTANCE_SETUP_CALLBACK`, é chamada para notificar um *minifilter* de que um *volume* está disponível e para configurar uma *instance* para esse *volume*. É possível

[25] Funções e estruturas da API de *minifilters* com nome `FltXXX` são funções que estão disponíveis para os componentes *kernel mode*, enquanto que funções com nome `FilterXXX` são funções que estão disponíveis para os componentes *user mode*.

especificar exatamente em quais tipos de *volumes* um *minifilter* está interessado, baseado no tipo de dispositivo do *volume* e no *file system* utilizado por ele.

Analogamente, a função `InstanceQueryTeardown`, que deve possuir assinatura compatível com a definida pelo tipo `PFLT_INSTANCE_QUERY_TEARDOWN_CALLBACK`, é chamada sempre que um *volume* no qual uma *instance* estava acoplada está sendo desmontado (*unmounted*), ou ainda quando o próprio *minifilter* está sendo descarregado. Ela é responsável por realizar o devido processamento quando da destruição (*teardown*) de uma *instance*. O *minifilter* pode ainda especificar na estrutura `FLT_REGISTRATION` callbacks, geralmente nomeadas `InstanceTeardownStartCallback` e `InstanceTeardownCompleteCallback`, e com assinatura definida pelo tipo `PFLT_INSTANCE_TEARDOWN_CALLBACK`, que são chamadas no início e fim desse processo de destruição da *instance*.

Um *minifilter* deve chamar, dentro da `DriverEntry`, após as devidas e relevantes configurações, a função `FltStartFiltering` para notificar ao *Filter Manager* que está pronto para filtrar as requisições de I/O que chegarem.

Uma vez em funcionamento, um *minifilter* pode ser explicitamente descarregado através do uso de uma requisição de *stop* de serviço (`net stop <filter_name>` ou `sc stop <filter_name>`) ou programaticamente, através de funções da API de *minifilters* (`FltUnloadFilter` e `FilterUnload`).

Quando um *minifilter* é descarregado, uma *callback*, especificada na estrutura `FLT_REGISTRATION` e geralmente chamada de `FilterUnloadCallback`, é chamada. Tal *callback* deve possuir assinatura compatível com a definida pelo tipo `PFLT_FILTER_UNLOAD_CALLBACK`. Ela deve fechar qualquer *communication port* (cujo conceito será explicado mais adiante) que esteja aberta, chamar a função `FltUnregisterFilter` para desregistrar o *minifilter* junto ao *Filter Manager*, e realizar qualquer tipo de limpeza (*cleanup*) necessária. Na verdade, o registro dessa *callback* é opcional, mas não especificá-la acarreta o ônus do *minifilter* não poder ser descarregado.

Para filtrar as requisições de I/O desejadas, o *minifilter* pode registrar, para cada tipo de requisição, duas *callbacks*: uma *preoperation callback*, que é executada **antes** do *file system driver* completar a operação de I/O, e uma *postoperation callback*, que é executada **depois** do *file system driver* completar a operação de I/O. Tais *callbacks* podem, além de realizar os devidos processamentos adicionais, completar antecipadamente ou cancelar (vetar) as

requisições de I/O. Para tal, a *callback* deve alterar adequadamente o campo `IoStatus` da estrutura `FLT_CALLBACK_DATA` recebida por ela como argumento.

O sistema distingue os tipos de requisição de I/O através de códigos de identificação do IRP. Para cada IRP, existem dois códigos: um *major function code*, que identifica a natureza geral daquela requisição, e um *minor function code*, que identifica detalhes específicos daquela requisição individual. Para construir o *minifilter*, apenas o *major function code* é necessário ser conhecido e especificado. A tabela a seguir sumariza os *major function codes* de maior interesse para *minifilters*[26]:

IRP major function code	Descrição
IRP_MJ_CLEANUP	Indica que o útlimo *handle* do *file* associado foi fechado (mas pode ainda não ter sido liberado)
IRP_MJ_CLOSE	Indica que o último *handle* do *file* associado foi fechado e liberado. Todas as requisições de I/O que restavam foram completadas ou canceladas
IRP_MJ_CREATE	Indica uma requisição para abrir um *handle* de um *file* ou *device*
IRP_MJ_READ	Indica uma requisição, por parte de outra aplicação (ou componente), de transferência (leitura) de dados a partir de um *device*
IRP_MJ_WRITE	Indica uma requisição, por parte de outra aplicação (ou componente), de transferência (escrita) de dados para um *device*

TAB. 4-2: IRP *major function codes* mais relevantes no desenvolvimento de um *minifilter*

Para o presente trabalho, estamos interessados nas requisições com *major function code* igual a `IRP_MJ_CREATE`, i.e., as requisições de abertura de arquivos, pois desejamos vetar a abertura de arquivos que tenham sido verificados como estando infectados.

Dentre as duas *callbacks* para filtrar as requisições, apenas a *preoperation callback* é obrigatória, sendo a *postoperation callback* opcional. As *preoperation callbacks* dos *minifilters* interessados em uma dada requisição serão chamadas na ordem direta das *instances*, determinada pelas suas *altitudes*. Por outro lado, as *postoperation callbacks* serão chamadas na ordem inversa determinada pelas *altitudes* das *instances*.

A *preoperation callback* deve realizar qualquer processamento que seja necessário antes da concretização da requisição de I/O. Através do seu valor de retorno,

[26] Para uma relação mais completa de IRP *major function codes*, consultar a relação constante em http://msdn.microsoft.com/en-us/library/ms806157.aspx.

ela indica se a requisição teve sucesso ou não, e se a *postoperation callback* deve ser chamada (código `FLT_PREOP_SUCCESS_WITH_CALLBACK`) ou não (código `FLT_PREOP_SUCCESS_NO_CALLBACK`).

A *postoperation callback* possibilita ao *minifilter* realizar qualquer tipo de processamento pós-concretização da requisição. É importante ressaltar que, no momento da execução dessa *callback*, o *minifilter* ainda pode vetar a requisição através da correta manipulação do campo `IoStatus` da estrutura `FLT_CALLBACK_DATA` recebida como argumento pela *callback*.

A comunicação de *minifilters* com componentes *user mode* é feita através do uso de *communication ports*.

Após criar uma *communication server port*, o *minifilter* espera que componentes *user mode* se conectem a ela. Quando um componente *user mode* tenta se conectar, através do uso da função `FilterConnectCommunicationPort`, uma *callback*, geralmente nomeada `ConnectNotifyCallback` e com assinatura determinada pelo tipo `PFLT_CONNECT_NOTIFY`, é chamada. Uma conexão só é aceita se o componente *user mode* tiver privilégio de acesso suficiente, segundo especificado pelo *security descriptor* associado à porta. Quando tal *callback* retorna, o componente *user mode* recebe um *handle* que representa o lado *user mode* da conexão. O componente *user mode* pode então usar esse *handle* para enviar informações de volta ao *minifilter*.

Fechar qualquer dos lados da comunicação, encerra a conexão. Se o componente *user mode* se desconectar, uma *callback* do *minifilter*, geralmente nomeada `DisconnectNotifyCallback` e com assinatura determinada pelo tipo `PFLT_DISCONNECT_NOTIFY`, é chamada.

Quando uma mensagem do componente *user mode*, enviada através do uso da função `FilterSendMessage`, chega ao *minifilter*, uma *callback*, geralmente nomeada `MessageNotifyCallback` e com assinatura determinada pelo tipo `PFLT_MESSAGE_NOTIFY`, é chamada.

Para enviar uma mensagem ao componente *user mode*, o *minifilter* pode fazer uso da função `FltSendMessage`. O componente *user mode* pode recuperar a mensagem chamando a função `FilterGetMessage`. Para responder uma mensagem enviada pelo *minifilter*, o componente *user mode* pode ainda usar a função `FilterReplyMessage`.

4.6. CONCEITOS E RECOMENDAÇÕES ACERCA DE PROGRAMAÇÃO EM KERNEL MODE

Agora definiremos alguns conceitos importantes para a programação em *kernel mode* para Windows e teceremos algumas recomendações acerca dela.

A primeira coisa a ser dita sobre a programação em *kernel mode* é que **ela se aproxima mais da realidade da máquina do que a programação em *user mode***. Isso significa que **o desenvolvedor vai precisar lidar com detalhes baixo nível com os quais antes, em uma aplicação *user mode*, ele não necessitava lidar**.

Como exemplo, temos a ocorrência de *page faults*. Para uma aplicação normal (*user mode*), uma *page fault* resulta em *swapping* para acessar a página faltante na memória. Todo o processo ocorre de maneira transparente para a aplicação, à qual parece, devido ao mecanismo de memória virtual, que todas as páginas que ela necessita encontram-se carregadas na memória principal e podem ser utilizadas diretamente. Para um componente em *kernel mode*, no entanto, a ocorrência de uma *page fault* em determinadas circunstâncias pode levar a um *system crash*.

Conceitos Importantes:

- Interrupções e IRQLs:

O mecanismo de interrupções é um mecanismo utilizado para permitir que o sistema operacional responda de maneira mais eficiente a eventos do *hardware*, geralmente eventos ligados a operações de I/O.

Quando ocorre uma interrupção, o processador deve interromper brevemente o que estava fazendo e executar uma rotina de tratamento de interrução adequada (*Interrupt Service Routine* – ISR).

Em relação ao sistema operacional, a interrupção não cria uma nova *thread*. Ao invés disso, o Windows interrompe a *thread* atual pelo curto período de tempo necessário para o tratamento da interrupção e executa, no contexto desta *thread*, a ISR, retornando após concluído esse processamento o controle da *thread* ao seu respectivo dono.

Uma interrupção pode ser também disparada por eventos de *software*. Essas interrupções de *software* são suportadas no Windows sob a forma de *deferred procedure calls* (DPCs), que são utilizadas por *drivers*, entre outras coisas, para lidar com os aspectos que consomem mais tempo do tratamento de interrupções de *hardware*.

Para garantir que certas interrupções mais importantes e com requisitos mais sensíveis de tempo sejam tratadas antes das demais, foi criado o conceito de Interrupt Request Level (IRQL).

A cada momento em que está processando alguma rotina, o processador roda sob uma determinada IRQL. Essa IRQL determina quais interrupções podem interromper a thread atual e quando elas serão tratadas. Basicamente, uma IRQL maior indica processamento com precedência sobre os demais. As próprias ISRs possuem uma IRQL associada ao tipo de interrupção que tratam, indicando, portanto, quais outras interrupções são mais importantes que ela e podem interromper seu tratamento.

Quando uma interrupção ocorre, o sistema compara a IRQL da interrupção ocorrida com a IRQL do processador atualmente.

Caso a IRQL da interrupção seja maior que a do processador, o sistema eleva a IRQL do processador ao nível da interrupção. O código que estava sendo executado é suspenso e não é retomado até que a ISR tenha terminado e a IRQL do processador tenha baixado de volta ao seu valor original. A ISR em questão pode, por sua vez, ser interrompida por outra ISR com IRQL maior.

Caso a IRQL seja igual a do processador, a ISR deve esperar que todas as rotinas anteriores com mesmo IRQL tenham terminado. Ela então executará até terminar, a menos que seja interrompida por alguma interrupção com IRQL maior.

Cabe destacar que a análise acima é válida para um ambiente monoprocessado. No tocante a ambientes, com mais de um processador, é possível possuir um processador rodando uma rotina com uma IRQL e outro com outra IRQL de valor distinto. Por isso, deve-se tomar cuidado com relação às suposições feitas com relação à concorrência e sincronização de código em *kernel mode*.

As IRQLs possuem, cada uma, um valor numérico associado, mas esse valor varia de acordo com a arquitetura do processador utilizado. Por isso, as IRQLs são geralmente referidas por um nome. A tabela a seguir sumariza as principais IRQLs de interesse no desenvolvimento de *drivers*[27]:

[27] Mais informações sobre IRQLs e seus valores podem ser obtidas em http://msdn.microsoft.com/en-us/library/ms810029.aspx.

Nome da IRQL	Valor da IRQL		Descrição
	x86	AMD64	
PASSIVE_LEVEL	0	0	Utilizado por todas as aplicações *user mode* e pela maioria das rotinas *kernel mode*.
APC_LEVEL	1	1	Utilizado por *asynchronous procedure calls* (APCs) e quando da ocorrência de *page faults*.
DISPATCH_LEVEL	2	2	Utilizado pelo escalonador de *threads* (*dispatcher*) e por *deferred procedure calls* (DPCs).
Device Interrupt Level (DIRQL)	3-26	3-11	Utilizado por interrupções de *devices* e suas ISRs.
HIGH_LEVEL	31	15	Relacionado a erros de máquina e erros catastróficos do sistema.

TAB. 4-3: Principais IRQLs de interesse no desenvolvimento de *drivers*

É importante notar que o próprio escalonador do sistema operacional roda em uma determinada IRQL, no caso a DISPATCH_LEVEL. **Isso significa que qualquer código que rode em uma IRQL maior ou igual a DISPATCH_LEVEL está efetivamente impedindo o sistema de escalonar qualquer outra *thread* para rodar. Qualquer erro ocorrido nesse nível pode, portanto, levar o sistema a um *system crash*, pois ele não poderá escalonar outras *threads* para cuidar do erro ocorrido.** Da mesma forma, **uma *page fault* que ocorra em DISPATCH_LEVEL não será tratada**.

Assim, sendo, deve-se prestar muito atenção à IRQL em que cada rotina de *kernel mode* pode (ou deve) rodar e as consequências disso para o sistema.

- Contextos de *threads*:

As aplicações *user mode* normalmente criam e controlam suas próprias *threads*. Por outro lado, rotinas de *kernel mode* normalmente não criam *threads* ou executam em *threads* especialmente criadas para seu uso.

Ao invés disso, tais rotinas rodam em uma *thread* arbitrária. O sistema "pega emprestado" a *thread* atualmente rodando no processador e a utiliza para executar a rotina em questão. Dessa forma, um *driver* que execute uma determinada rotina de tratamento de uma requisição de I/O pode estar rodando tanto no contexto da *thread* que iniciou a requisição em questão como também no contexto de qualquer outra *thread* do sistema.

Dessa forma, **nenhuma associação explícita deve ser feita no *driver* entre a *thread* que originou uma requisição e a rotina que deve tratar essa requisição**. Além disso, em um sistema com mais de um processador, **é possível ter uma mesma rotina sendo executada simultaneamente em dois ou mais processadores, em contextos de *threads* diferentes**.

- Gerenciamento de memória:

Devido ao fato de todos os componentes em *kernel mode* compartilharem o mesmo espaço de memória, os *drivers* devem ter extremo cuidado no tocante ao gerenciamento da memória. Não existe nenhum mecanismo de proteção contra acesso indevido de memória, como o que existe em *user mode*. **Qualquer erro que provoque a corrupção da memória, provocará também um *system crash***. Da mesma forma, a aplicação não deve abusar da alocação de memória, pois, como se trata de um recurso compartilhado, acabará por prejudicar os demais componentes *kernel mode* e, por conseguinte, o próprio sistema como um todo, podendo levar inclusive a sua falha completa pela falta de memória para realizar operações mais básicas e vitais.

Podemos dividir o gerenciamento de memória em 3 áreas: gerenciamento de *page faults*, alocação dinâmica e uso da pilha. No tocante ao gerenciamento de *page faults*, devemos, conforme foi dito anteriormente atentar para o fato de que, dependendo do nível IRQL atual, uma *page fault* que venha a ocorrer não necessariamente poderá ser tratada. Rotinas que executem em uma IRQL maior ou igual a DISPATCH_LEVEL levarão um sistema a um *system crash* caso ocorra uma *page fault*.

No tocante à alocação dinâmica de memória, temos em *kernel mode* não um, mas sim 2 tipos de *heaps* para alocação de memória: o *paged pool* e o *nonpaged pool*. O *paged pool* contém memória que pode ser paginada para o disco rígido, caso seja necessário. O *nonpaged pool* contém memória que não pode ser paginada e deve estar sempre residente na memória principal. Obviamente, o *paged pool* não deve ser usado em situações com IRQL maior ou igual a DISPATCH_LEVEL, pois uma *page fault* no acesso a essa memória seria nesse caso desastrosa. Nesse caso, deve-se usar o *nonpaged pool*. No entanto, ele é um recurso limitado que deve ser usado com muita cautela e parcimônia.

No tocante ao uso da pilha, temos que ela também é um recurso muito caro e escasso. Dessa forma, os componentes *kernel mode* não devem abusar de chamadas recursivas e de estruturas de dados gigantescas como argumentos para executarem suas tarefas, procurando

realizá-las de forma mais simples e iterativa, com o menor uso de memória possível, de forma a não prejudicar os demais componentes *kernel mode* do sistema.

Recomendações:

- **Prepare o *driver* para lidar com condições de baixa quantidade de memória disponível:** o *driver* deve estar preparada para condições em que, pela falta de memória disponível, não posssa alocar estruturas de dados necessárias para sua operação. Ele não deve falhar desastrosamente nessas situações ou tentar acessar regiões de memória para as quais não tenha permissão de acesso.
- **Aloque a memória a partir do *pool* adequado:** não aloque indiscriminadamente a memória do *nonpaged pool*. Identifique corretamente as suas necessidades e aloque a memória a partir do *pool* adequado, conforme a IRQL utilizada na sua função.
- **Não confie em nada que vem de aplicações *user mode*:** sempre valide todos os dados que vêm de aplicações *user mode*. Verifique que o tamannho dos *buffers* está correto e que seus dados são válidos e não foram adulterados ou corrompidos.
- **Não simplesmente acesse os dados na posição referenciada por um ponteiro *user mode* em uma rotina *kernel mode*:** lembre-se que um ponteiro *user mode* só tem sentido no contexto do processo original do qual ele foi obtido. Para acessar corretamente os dados apontados por um ponteiro *user mode*, o *driver* deve primeiro validar o ponteiro através do uso da rotina `MmProbeAndLockPages`, ou as macros `ProbeForRead` e `ProbeForWrite`. Com isso, o sistema acessa a real página da memória onde se encontra o dado referenciado pelo ponteiro, permitindo a correta obtenção do dado alvo. Todos os acessos a ponteiros *user mode* devem ser feitos dentro de construtos *try-except*, de forma a proteger o *driver* de qualquer erro que possa ocorrer caso o ponteiro venha a ser invalidado enquanto se tenta acessá-lo.
- **Tenha plena consciência dos requisitos de IRQL que as rotinas *kernel mode* têm e das implicações destes requisitos:** cada rotina *kernel mode* pode ter requisitos específicos de rodar em um dado nível IRQL. É de fundamental importância conhecer tais requisitos. Dependendo de sob qual IRQL uma rotina roda, a abordagem para sincronização, gerenciamento de memória e de recursos compartilhados do sistema terá de ser diferente. Interpretar erradamente sob qual IRQL uma rotina roda (ou pode rodar) poderá levar a

diversos problemas graves, como *deadlocks*, corrupção de memória e até *system crashes*. A API especifica claramente quais os requisitos de cada rotina provida por ela.

- **Tenha muito cuidado com rotinas que levam muito tempo para ser concluídas:** deve-se ter muito cuidado com rotinas que levam muito tempo para ser concluídas, pois, caso essa rotina seja exaustivamente chamada pelos mais diversos motivos, o desempenho do sistema pode ser severamente degradado. Além disso, cabe lembrar que uma rotina que rode sobre um nível IRQL maior ou igual a DISPATCH_LEVEL está efetivamente bloqueando o sistema de escalonar outra *thread*. Caso seja necessário executar algum tipo de processamento pesado e complementar, os *drivers* devem criar *work items*, estruturas de dados que especificam tarefas, e que devem ser colocadas em uma fila (*system work queue*) para posterior processamento pelo sistema.

5. TÓPICOS UTILIZADOS DA API DO WINDOWS

5.1. CONCEITO E ORGANIZAÇÃO DA API

Para o desenvolvimento do protótipo do anti-vírus, é necessário o uso de diversas funções que interagem diretamente com o sistema operacional Windows em *User Mode*, como funções de controle do registro do Windows, de envio e tratamento de *Window Messages*, operações com DLLs, etc.

O conjunto das funções disponibilizadas pela Microsoft para a realização dessa interação com o sistema operacional em alto nível é denominado **API**[28] **do Windows** ou informalmente **WinAPI**.

Nesse capítulo, iremos estudar algumas dessas funções da API do Windows que foram necessárias para o desenvolvimento do protótipo. Não iremos estudar todas as funções, porque a API do Windows possui milhares de funções e muitas delas não são úteis para este projeto, como as funções que interagem com a rede.

Historicamente, a API do Windows foi criada em 1985 juntamente com o lançamento da primeira versão do Windows, o Windows 1.0, que nem foi tão anunciado ao mundo.

Esse Windows era muito rudimentar na época, não possuindo divisão da memória principal em *kernel mode* e *user mode* e operando em modo real do processador. Conseqüentemente, a primeira versão da API do Windows também era bem rudimentar compondo apenas de 450 funções. Muitos desenvolvedores que programavam na época usando a WinAPI relatam que ela era muito mal documentada e não era segura, podendo-se realizar acidentalmente determinadas operações que causavam vários *crashs* no Windows.

Ao longo dos anos, à medida que o Windows também evoluía, a API também melhorava, tornando-se mais completa e segura. Durante todo o tempo que houve a evolução, os desenvolvedores da Microsoft se preocuparam em manter a compatibilidade com as versões antigas, mantendo as funções antigas nas versões novas.

De fato, hoje em dia, pode-se notar que existem muitas funções na API terminadas em Ex, significando *Extended*, que correspondem a funções antigas da API, mas que possuem uma funcionalidade nova (a função `SleepEx` corresponde a função `Sleep`).

O ponto crucial em que houve uma alteração drástica na API do Windows foi em 1990 com o lançamento do Windows 3.0, o primeiro que teve repercussão mundial. Esse o

[28] API é a sigla de *Application Programming Interface*

Windows era o primeiro a ser de 32-bits, ao contrário dos Windows 1.0 e 2.0 que eram de 16-bits, e como tal sua API precisou ser migrada para a então nova plataforma de 32-bits.

Atualmente a API do Windows possui milhares de funções. Basicamente, ela é divida por áreas de abrangência das funções. Cada área dessa de abrangência é implementada por uma biblioteca de linkagem dinâmica (DLL) diferente. São as áreas:

- **Serviços básicos**: compreende as funções que lidam o acesso a sistemas de arquivos, dispositivos, processos, *threads* e controle de erro. Essas funções são implementadas na biblioteca **kernel32.dll**.

- **Serviços avançados**: compreende as funções que interagem com o kernel, operando no Registro do Windows ou fornecendo serviços como desligamento e reiniciação do computador. Implementada pela biblioteca **advapi32.dll**.

- **GDI (*Graphical Device Interface*)**: compreende as funções relacionadas a parte gráfica do Windows como criação de janelas padronizadas, etc. Implementada pela biblioteca **gdi32.dll**.

- **Interface com o usuário**: funções relacionadas a parte gráfica, mas que ao contrário da GDI que cuida da padronização, essa área cuida das funções que fazem a interação com o usuário, como uso de *Windows Message* (que será explicado mais tarde), entradas do teclado e entradas do mouse. Tais funções são implementadas pela biblioteca **user32.dll**.

- **Serviços de Rede**: funções relacionadas a parte de comunicação de rede, como DNS, DHCP, etc. Muitas dessas funções são implementadas por bibliotecas de terceiros, não existindo uma padronização de implementação (apenas da interface).

Nas próximas seções serão mostrados alguns dos tópicos da API que foram utilizados no desenvolvimento do protótipo do anti-vírus. Antes, na seção posterior a essa, será apresentado algumas considerações gerais que devem ser tomadas quando se trabalha com a WinAPI.

5.2. CONSIDERAÇÕES INICIAIS

Quando um programador acostumado com o desenvolvimento de programas em C/C++ se depara pela primeira vez com a API do Windows, ele se surpreende com um mundo aparentemente completamente diferente do ambiente normal de programação a que ele está acostumado.

A API do Windows possui uma série de palavras reservadas definidas que são completamente diferentes da que se usa na programação normal. Palavras como HANDLE, WINAPI, __IN, ___OUT, LRESULT, LPCSTR, LPWSTR, HWND, BYTE, WORD,

DWORD, etc. Muitas dessas palavras reservadas são, na verdade, maneiras alternativas de se definir alguns tipos primitivos.

A primeira coisa exótica que se depara quando se começa a programar para Windows é o ***handle***. Diversas funções da WinAPI operam diretamente com *handles*. Assim é indispensável explicar esse elemento para completo entendimento das seções posteriores.

O *handle* nada mais é que um indentificador único para um recurso do sistema operacional Windows. Esse recurso pode ser uma área de memória compartilhada, uma pesquisa de diretório, uma chave do Registro do Windows aberta, etc.

O livro "*Windows Internals*" (SOLOMON e RUSSINOVICH 2005) faz uma breve explanação sobre a definição de *handle* na página 878: "*An object indentifier. A process receives a handle to an object when it creates or opens an object by name. Referring to an object by its handle is faster than using its name because the object manager can skip the name lookup and find the object directly.*".

Assim, se o acesso a um objeto como uma chave do registro é mais rápido usando o *handle* do que por exemplo o nome completo da chave, então vê-se que é indispensável o uso de *handle* por parte de qualquer função de manipulação de objetos.

Quando analisamos uma função da WinAPI freqüentemente nos deparamos com diversas palavras reservadas e estranhamos pelo fato de nos acostumarmos a pensar que elas são modificadores para a função. Na verdade, muitas das palavras reservadas são apenas informações adicionais a cerca dos parâmetros passados, não possuindo nenhum valor para efeitos de compilação.

Tomemos o exemplo da função `FormatMessage`[29], que permite traduzir o código de erro para um formato de mensagem mais amigável ao usuário. Se consutarmos a API do Windows seu cabeçalho é o seguinte:

```
DWORD WINAPI FormatMessage(
  __in      DWORD dwFlags,
  __in_opt  LPCVOID lpSource,
  __in      DWORD dwMessageId,
  __in      DWORD dwLanguageId,
  __out     LPTSTR lpBuffer,
  __in      DWORD nSize,
  __in_opt  va_list *Arguments
);
```

[29] http://msdn.microsoft.com/en-us/library/ms679351(VS.85).aspx

Analisando este cabeçalho, conseguimos distinguir pelo menos quatro palavras reservadas que, embora pareça, não são modificadores de função. São elas: `WINAPI`, `__in` , `__in_opt` e `__out`.

A palavra reservada `WINAPI` possui a finalidade de indicar que uma determinada função pertence a API do Windows.

As palavras reservadas , `__in` , `__in_opt` e `__out`, juntamente com `__out_opt`, possuem a finalidade de classificar um argumento de uma função quanto o seu emprego dentro da função e quanto a sua obrigatoriedade.

As palavras `__in` e `__in_opt` servem para indicar que este argumento é um parâmetro de entrada da função, i.e., que a função utilizará o dado passado para seu correto funcionamento. A diferença entre ambos é que o primeiro é obrigatório e o segundo não. Caso não se deseje passar um parâmetro não-obrigatório, geralmente passa-se NULL ou 0 no seu lugar, dependendo do tipo do parâmetro. Em caso de não se passar o parâmetro, a função provavelmente utilizará um valor padrão no lugar do dado passado.

As palavras `__out` e `__out_opt` servem para indicar que este argumento é um parâmetro de saída da função, i.e., que ele armazenará um valor que será escrito no decorrer da função. Alguns parâmetros de saída podem ser opcionais, pois em alguns momentos não se deseja armazenar um determinado dado.

Além desses 4 tipos, existem também as palavras `__in_out e __in_out_opt`, usadas em alguns casos quando o argumento acumula as funções de parâmetro de entrada e saída.

Conforme visto no cabeçalho da função apresentada, existem diversas palavras reservadas, que são modificadores de função ou de tipos, mas que aparentemente são novos, como, por exemplo, o tipo `DWORD`[30].

Na verdade, este tipo não é um tipo diferente, mas outra maneira de chamar um tipo primitivo. Se olharmos a referência a este tipo no arquivo `windef.h`, encontramos a seguinte linha:

```
typedef unsigned long   DWORD;
```

Mas por que se faz esta renomeação deste tipo primitivo ?

Por dois motivos. O primeiro é para reduzir o nome do tipo, já que `unsigned long` é um nome muito maior que `DWORD`. O segundo é que, apesar de o nome `unsigned long` ser

[30] `BYTE`: valor de 8 bits
`WORD`: valor de 16 bits
`DWORD`: valor de 32 bits
`QWORD`: valor de 64 bits

muito mais familiar ao programador comum, o nome `DWORD` é bem mais familiar para o programador que usa a WinAPI, pois, quando se lida com operações de mais baixo nível como a interação com o sistema operacional, este nome é muito comum.

Ao analisar os diversos tipos de dados no Windows, pode-se perceber que para muitos tipos de dados, existe um tipo correspondente precedido de P ou LP. Por exemplo, além do tipo `DWORD`, encontramos os tipos `PDWORD` e `LPDWORD`.

Os prefixos P e LP são padrões da nomenclatura da WinAPI e significam respectivamente *pointer* e *long pointer*, ou seja, indicam ponteiros para o tipo definido pelas outras letras da palavra reservada. Assim, `LPDWORD` é um ponteiro longo para `DWORD`, assim como `LPBYTE` é um ponteiro longo para `BYTE`, etc.

Uma última consideração sobre a API do Windows é a atenção que se deve ter entre a diferença entre **char** e **wide char**. Uma olhada mais cuidadosa na WinAPI, mostrará que existem esses dois tipos de caractere e conseqüentemente dois tipos de strings, uma para cada tipo de caractere. O primeiro tipo de caractere, o char, é o relativo ao padrão **ANSI**, enquanto o segundo é relativo ao padrão **Unicode**.

O padrão Ansi de caracteres é o padrão ASCII comum em programação em que se define um código de 8 bits que relaciona um número a um caracter padrão do mundo ocidental.

O padrão Unicode de caracteres foi definido em 1991 e corresponde a um código de 16 bits, ao contrário do padrão ANSI, que possui 8 bits. O motivo da extensão do tamanho do caractere foi para incorporar no código, muitos caracteres do mundo oriental que não podia ser incorporados no padrão ANSI devido a insuficiência de espaço para armazenamento. Com 16 bits, é em tese perfeitamente possível representar todos os caracteres de todas as culturas, inclusive do mandarim chinês aos caracteres árabes.

A partir dos Windows de família NT, a Microsoft passou a adotar em suas aplicações o padrão Unicode[31]. Conseqüentemente, a API do Windows passou a dar suporte a esse tipo de caractere Unicode.

Foi criada, dessa maneira, o wide char e a wide string e para efeitos de compatiblização toda a função da API do Windows que opere com char ou string, possui duas versões uma para ANSI, terminada com A, e outra para Unicode, terminada em W.

[31] Embora não tenha sido parte do projeto inicial a implementação do padrão Unicode, para estender este padrão aos Windows da família 9X (95, 98 e Me) foi criada a Microsoft Layer *for Unicode* (MSLU).

Assim tomemos, por exemplo, a função `GetCurrentDirectory`[32] da WinAPI, que obtém o caminho completo do diretório corrente do processo que está sendo rodado no momento.

Essa função, como possui uma string usada como argumento passado para a função para armazenamento do diretório, possui duas versões implementadas `GetCurrentDirectoryA`, para strings ANSI, e `GetCurrentDirectoryW`, para Unicode. Seus respectivos cabeçalhos são descritos abaixo:

```
DWORD WINAPI GetCurrentDirectoryA(
  __in   DWORD nBufferLength,
  __out  LPCSTR lpBuffer
);

DWORD WINAPI GetCurrentDirectoryW(
  __in   DWORD nBufferLength,
  __out  LPWSTR lpBuffer
);
```

Como se pode notar a diferença entre ambas as funções é que na primeira a variável `lpBuffer` é do tipo `LPCSTR`, ou seja, é um ponteiro para char; enquanto na segunda ela é do tipo `LPWSTR`, ou seja, um ponteiro para wide char.

Apesar de ser sutil essa diferença, deve-se ter muita atenção quanto a isso. Às vezes, estamos operando com char ANSI e, quando utilizamos uma função dessas, esquecemos de adicionar o sufixo A. Como na maioria das plataformas Windows, por padrão, quando omitimos o sufixo, a WinAPI entende que queremos nos referir ao padrão Unicode, os resultados que esta operação pode causar são os mais desastrosos, sendo às vezes muito difíceis de serem reconhecidos.

5.3. ESBOÇO DE UM PROGRAMA PARA WINDOWS

A estrutura de um programa escrito para ambiente Windows é completamente diferente da estrutura de um programa escrito para console (DOS ou Linux). Nessa seção abordaremos como é a estrutura de um programa Windows, mostrando quais são os passos necessários para construir esse esboço.

[32] http://msdn.microsoft.com/en-us/library/aa364934(VS.85).aspx

A primeira grande diferença entre um programa escrito para console e outro para Windows é a função principal, que é o ponto de entrada para o programa.

Na programação para console, a função principal atende pelo nome de `main` recebendo como argumentos respectivamente o número de parâmetros passados ao programa e um ponteiro para um *array* de ponteiros para as várias strings que contém os argumentos passados.

```
int main(int argc, char **argv);
```

Já na programação para Windows, a função principal, ponto de entrada do programa, é a função `WinMain`[33], que recebe diversos parâmetros, conforme mostra o cabeçalho abaixo:

```
int WINAPI WinMain(
    HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow
);
```

Primeiramente, o parâmetro `hInstance` recebido por esta função representa o *handle* atual que a aplicação recebe no início do programa.

Em seguida, temos o parâmetro `hPrevInstance` é um campo obsoleto, que não possui muita utilidade, sempre sendo nulo.

O parâmetro `lpCmdLine`, por sua vez, é um ponteiro para uma string contendo a linha de comando de execução desta aplicação (contendo obviamente os argumentos passados ao programa).

Finalmente, o parâmetro `nCmdShow` especifica o estado da janela principal a ser mostrada, podendo a tela iniciar minimizada, maximizada, oculta, etc.

Ao iniciar um programa para Windows a primeira coisa que se deve fazer é registrar a classe da janela principal do programa. Isso é feito utilizando a função `RegisterClass`[34], cujo cabeçalho é descrito abaixo:

```
ATOM RegisterClass(CONST WNDCLASS *lpWndClass);
```

Esta função recebe como parâmetro um ponteiro para uma estrutura `WNDCLASS`, que representa uma janela do Windows, e retorna um valor diferente de nulo em caso de sucesso.

A estrutura `WNDCLASS`[35] é definida da seguinte forma:

---

[33] http://msdn.microsoft.com/en-us/library/ms633559(VS.85).aspx
[34] http://msdn.microsoft.com/en-us/library/ms633586(VS.85).aspx
[35] http://msdn.microsoft.com/en-us/library/ms633576(VS.85).aspx

```
typedef struct {
    UINT style;
    WNDPROC lpfnWndProc;
    int cbClsExtra;
    int cbWndExtra;
    HINSTANCE hInstance;
    HICON hIcon;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCTSTR lpszMenuName;
    LPCTSTR lpszClassName;
} WNDCLASS, *PWNDCLASS;
```

Nessa estrutura, podemos distinguir diversos parâmetros que devem ser passados para a mesma para correto registro da tela. A tabela a seguir apresenta os elementos que compõe esta estrutura e seus respectivos significados:

Parâmetro	Significado
`style`	Inteiro indicando o estilo da classe
`lpfnWndProc`	Ponteiro para função que tratará as *Windows Messages*
`cbClsExtra`	Reservado (preenche-se com 0)
`cbWndExtra`	Reservado (preenche-se com 0)
`hInstance`	*Handle* da aplicação Windows que contém essa janela
`hIcon`	*Handle* para o ícone dessa janela
`hCursor`	*Handle* para o tipo de cursor dessa janela
`hbrBackground`	*Handle* para a cor do fundo da janela
`lpszMenuName`	Menu da janela
`lpszClassName`	Nome da janela

TAB. 5-1: Elementos que compõe a estrutura WNDCLASS e seus respectivos significados

Após o registro da janela com sucesso, deve-se criar a janela, o que é feito utilizando a função `CreateWindow`[36]. O cabeçalho dessa função é descito abaixo:

[36] http://msdn.microsoft.com/en-us/library/ms632679(VS.85).aspx

```
HWND CreateWindow(
    LPCTSTR lpClassName,
    LPCTSTR lpWindowName,
    DWORD dwStyle,
    int x,
    int y,
    int nWidth,
    int nHeight,
    HWND hWndParent,
    HMENU hMenu,
    HINSTANCE hInstance,
    LPVOID lpParam
);
```

O retorno dessa função é um *handle* que identifica a janela. Esse *handle* é muito precioso para o envio de *Windows Messages*, conforme será visto na próxima seção. Caso a criação da janela falhe, a função retornará `NULL`, ao invés desse *handle*.

Quanto aos argumentos passados, eles são explicados na tabela abaixo:

Parâmetro	Significado
`lpClassName`	Nome da classe da janela
`lpWindowName`	Nome da janela que aparecerá na barra de título
`dwStyle`	Estilo da janela
`x, y`	Posição inicial da janela horizontal e vertical
`nWidth`	Largura da janela
`nHeight`	Comprimento da janela
`hWndParent`	*Handle* para a janela mãe a que essa janela pertence. Se for ao Desktop, deve-se passar `HWND_DESKTOP`.
`hMenu`	*Handle* para o menu do janela
`hInstance`	*Handle* da aplicação Windows
`lpParam`	Parâmetro de criação (geralmente `NULL`).

TAB. 5-2: Argumentos da função CREATEWINDOW e seus respectivos significados

Após a criação da janela com sucesso, deve-se exibi-la utilizando-se as funções `ShowWindow`[37] e `UpdateWindow`[38] nessa ordem.

```
BOOL ShowWindow(HWND hWnd, int nCmdShow);
BOOL UpdateWindow(HWND hWnd);
```

A primeira função faz com que a janela seja mostrada na tela. Ela recebe dois parâmetros: o primeiro é o *handle* da janela a ser mostrada e o segundo, o estado da janela a ser mostrada. Caso a janela mostrada esteja num estado visível, a função retorna um valor diferente de zero. Caso contrário, ela retorna a zero.

A segunda função pinta a janela, atualizando-a visualmente. Recebe como parâmetro o *handle* da janela. Em caso de sucesso, a função retorna um valor diferente de zero.

Após todo esse procedimento, a aplicação Windows criou uma janela e está pronto para que esta janela possa receber *Windows Messages*, que são mensagens que permitem as janelas interagirem com o usuário.

A próxima seção explicará melhor o que elas são, como é seu funcionamento e como manipulá-las para realizar a comunicação entre processos.

5.4. *WINDOWS MESSAGING*

As *Window Messages* são a forma mais comum de comunicação entre aplicações Windows que possuam janelas. São utilizadas no tratamento de eventos simples como o clique do botão direito do mouse pelas janelas, o aperto de um botão ou a abertura de um menu.

De forma semelhante aos sinais utilizados nos sistemas UNIX, trata-se de mensagens enviadas pelo sistema operacional em forma de objetos a uma fila de mensagens, denominada *Message Queue*, presente em toda aplicação, sendo que esta se encarrega da retirada e tratamento dessas mensagens.

Uma *Window Message* é definida de acordo com a seguinte estrutura de dados:

[37] http://msdn.microsoft.com/en-us/library/ms633548.aspx
[38] http://msdn.microsoft.com/en-us/library/ms534874(VS.85).aspx

```
typedef struct {
  HWND hwnd;
  UINT message;
  WPARAM wParam;
  LPARAM lParam;
  DWORD time;
  POINT pt;
} MSG, *PMSG;
```

A tabela abaixo mostra o significado dos elementos que compõe essa estrutura:

Parâmetro	Significado
`hwnd`	*Handle* da janela que recebe esta mensagem
`message`	Inteiro sem sinal contendo o tipo de mensagem
`wParam, lParam`	Parâmetros da mensagem, i.e., informações adicionais a cerca da mensagem.
`time`	Instante em que a mensagem foi postada
`pt`	Posição do cursor do mouse no momento em que a mensagem foi postada

TAB. 5-3: Elementos que compõe a estrutura MSG e seus respectivos significados

Desta maneira, diversos eventos são tratados pelas janelas das aplicações Windows. Se, por exemplo, ocorre um duplo-clique com o botão direito do mouse na área de uma janela de uma determinada aplicação, esta recebe uma mensagem do tipo `WM_RBUTTONDBCLK` definido no item `message` da estrutura da mensagem e com os parâmetros `lParam` e `wParam` recebendo as coordenadas do cursor no momento do duplo-clique.

Os valores dos tipos de mensagens são organizados segundo faixas de valores. A tabela abaixo mostra como é essa organização:

Faixa de valores	Finalidade
0x0000 a 0x0400	Valores reservados pelo sistema, não podendo ser usados
0x0401 a 0x7FFF	Valores já definidos, utilizados pelas aplicações para o tratamento de eventos das janelas como abertura de menus, clique do mouse, etc
0x8000 a 0xBFFF	Valores utilizados pela aplicação em eventos que não são especificamente de janelas, como a notificação do término de uma sessão de usuário
0xC000 a 0xFFFF	Valores utilizados pelas aplicações, sendo reservados em tempo de execução de programas, através de rotina de registro de mensagens
Acima de 0xFFFF	Valores reservados pelo sistema, também não podendo ser utilizados

TAB. 5-4: Faixa de valores de tipos de *Windows Messages* e suas respectivas finalidades

Como pode ser notada, a penúltima faixa, de 0xC000 a 0xFFFF, pode ser usada para criação de mensagens personalizadas e, desta maneira, ser permitido a comunicação entre processos deste que estes sejam aplicações Windows que possuam janelas.

Para reservar um destes valores nesta faixa, para a criação de uma mensagem própria, utiliza-se a seguinte função da API do Windows[39] definida no arquivo `winuser.h` e implementada na biblioteca `user32.dll`:

```
UINT RegisterWindowMessageA(LPCSTR lpString);
```

Sendo que a função exige um parâmetro `LPCSTR`, que na verdade é uma string ANSI (*array* de `char`) com o nome da mensagem e retorna um inteiro sem sinal como um identificador único da mensagem.

Desta maneira, se duas aplicações distintas utilizarem a função passando o mesmo nome de mensagem, ambas receberão o mesmo número de identificação do tipo de mensagem, viabilizando o inicio da comunicação entre as duas. De posse deste número, uma pode enviar uma mensagem deste tipo para todas as janelas (*Broadcast*), com o *handle* da sua janela. A outra aplicação, que entenderá aquela mensagem, armazenará esse *handle* e mandará o *hanlde* da sua janela com a mesma mensagem personalizada, permitindo a primeira conhecer o seu *handle*. Com esses identificadores de janela trocados por ambas as aplicações, a comunicação entre os processos foi estabelecida.

[39] http://msdn.microsoft.com/en-us/library/ms644947.aspx

O envio de *Window Message* se faz basicamente duas funções e algumas outras derivadas destas, sendo que todas são descritas no arquivo `winuser.h` e implementadas na biblioteca `user32.dll`.

A primeira função é a `SendMessage`[40], cujo cabeçalho é definido da seguinte maneira:

```
LRESULT SendMessage(HWND hwnd, UINT Msg, WPARAM wParam, LPARAM lParam);
```

Onde `hwnd` é o *handle* da janela para qual será enviada a mensagem, `Msg` é o identificador do tipo de mensagem e `wParam` e `lParam` são os parâmetros da mensagem. O retorno da função é um inteiro longo, cujo significado varia de acordo com o tipo de mensagem passada.

Esta função envia a mensagem e espera o destinatário processar a mensagem para depois retornar. Dessa maneira, se não houver garantia de que o destinatário processe a mensagem ou mesmo a receba, esta função pode fazer a aplicação travar.

A segunda função é a `PostMessage`[41], cujo cabeçalho é:

```
LRESULT PostMessage(HWND hwnd, UINT Msg, WPARAM wParam, LPARAM lParam);
```

Ao contrário da primeira, esta não espera o destinatário receber e processar a mensage, retornando assim que a mensagem é enviada. É recomendada, portanto quando não se tem certeza de que haverá alguma resposta por parte do destinatário da mensagem.

Uma função alternativa para contornar este problema é a função `SendMessageTimeout`[42] derivada da `SendMessage`, que espera a mensagem ser processada para retornar ou, em caso de ausência de resposta, retorna depois de um período de tempo definido num dos seus parâmetros. Seu cabeçalho é o seguinte:

```
LRESULT SendMessageTimeout(HWND hwnd, UINT Msg, WPARAM wParam, LPARAM lParam, UINT fuFlags, UINT uTimeout, PDWORD_PTR lpdwResult);
```

Onde os quatros primeiros parâmetros são iguais a de `SendMessage`. O quinto parâmetro, `fuFlags`, trata-se das *flags* que configuram o modo como a *thread* em que a função está rodando se comportará ao receber novas mensagens para serem processadas. O parâmetro `uTimeout` é um inteiro sem sinal determinando o tempo de espera para a função retornar. O `lpdwResult`, por sua vez, representa o retorno que a função retornaria, caso fosse chamado `SendMessage`.

---

40 http://msdn.microsoft.com/en-us/library/ms644950(VS.85).aspx
41 http://msdn.microsoft.com/en-us/library/ms644944(VS.85).aspx
42 http://msdn.microsoft.com/en-us/library/ms644952(VS.85).aspx

Esta função retorna um valor diferente de zero em caso de retorno normal da função, isto é, em caso de a mensagem ser processada normalmente pelo destinatário; ou retorna zero em caso de erro ou de ser atingido o prazo de tempo de espera para retorno.

O tratamento de mensagens é realizado em duas etapas. A primeira etapa é a da manipulação da fila de mensagens (*Message Queue*), enquanto a segunda é a do tratamento das mensagens, uma a uma, propriamente dito.

Em C, o tratamento das mensagens é realizado basicamente por três funções: `GetMessage`[43], `TranslateMessage`[44] e `DispatchMessage`[45], sendo todas descritas no mesmo arquivo e implementada nas mesma biblioteca descritos anteriormente. O código em C responsável pela primeira parte é o seguinte:

```
BOOL ret;
// Loop do processamento da fila de mensagens
while( (ret = GetMessage( &msg, hWnd, 0, 0 )) != 0)
{
    if (ret == -1)
    { // em caso de haver erro
    }
    else
    { // Processamento normal das mensagens (sem WM_QUIT)
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

A função `GetMessage` retira a mensagem mais ao fim da fila de mensagem da *thread* em que é chamada. Possui o seguinte cabeçalho:

```
BOOL GetMessage(LPMSG lpMsg, HWND hWnd, UINT wMsgFilterMin, UINT wMsgFilterMax);
```

O primeiro cabeçalho, `lpMsg`, é um ponteiro para o endereço de uma estrutura MSG, que receberá a mensagem retirada da fila; o segundo parâmetro, `hWnd`, é um *handle* para janela que tratará esta mensagem; o terceiro e o quarto definem uma faixa de valores do tipo de mensagem que servirá como filtro para a retirada de mensagens.

[43] http://msdn.microsoft.com/en-us/library/ms644936(VS.85).aspx
[44] http://msdn.microsoft.com/en-us/library/ms644955(VS.85).aspx
[45] http://msdn.microsoft.com/en-us/library/ms644934(VS.85).aspx

Esta função retorna -1 em caso de erro, zero se a função retirar uma mensagem `WM_QUIT` ou um valor acima de zero, em caso de retirada de outro tipo de mensagem.

As funções `TranslateMessage` e `DispatchMessage` são responsáveis pela tradução e envio de mensagens para a função que processará cada mensagem uma a uma da janela, chamada de *Window Procedure*. O cabeçalho destas funções é o seguinte:

```
BOOL TranslateMessage(CONST MSG *lpMsg);
BOOL DispatchMessage(CONST MSG *lpMsg);
```

Onde `lpMesg` é um ponteiro para o endereço da estrutura `MSG` que armazena a mensagem.

A segunda é etapa é a codificação desta *Window Procedure*. Quando uma janela é criada em uma aplicação, é necessário registrar uma classe associada a esta janela e, na estrutura de dados associada a essa classe de janela, é passado um ponteiro para uma função *WindowProcedure* que fará o tratamento destas mensagens. Esta função deverá possuir a seguinte assinatura:

```
LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam,
LPARAM lParam);
```

Onde os parâmetros recebidos por esta função são exatamente os parâmetros da mensagem, isto é, *handle* da janela que recebe a mensagem, tipo de mensagem e primeiro e segundo parâmetros auxiliares da mensagem, respectivamente.

Nesta função deve ser tratada as mensagens que normalmente não são tratadas pela aplicação, como as mensagens personalizadas definidas com `RegisterWindowMessage`. Para todas as outras mensagens, deve ser utilizada a função `DefWinProc`, de mesma assinatura. Assim, o esboço do código desta função ficaria da seguinte maneira:

```
   LRESULT CALLBACK WindowProc(HWND hwnd, UINT uMsg, WPARAM wParam,
LPARAM lParam) {
        switch (uMsg)
        {
            case WM_ALGUMA_MESAGEM:
                // Código de resposta a esta mensagem
                return 0;
            //
            // Processamento de outras mensagens com outros "cases"
            //
            default:
                return DefWindowProc(hwnd, uMsg, wParam, lParam);
        }
        return 0;
    }
```

Em C#, essas duas etapas já estão pré-construídas quando uma janela (*Form*) é criada. Para adicionar o tratamento de novas mensagens, como mensagens personalizadas, a classe `Form` implementa um método denominado `WndProc` que pode ser sobrescrito por suas classes-filhas.

Este método recebe como parâmetro por referência um objeto da classe `Message`, contendo todas aquelas informações inerentes a estrutura MSG do C (`hWnd`, `uMsg`, `lParam` e `wParam`), permitindo o tratamento de mensagens da mesma maneira que as *Window Procedure* do C.

Assim como é necessário em C chamar a função `DefWindowProc` para dar um tratamento as mensagens não relacionadas, em C# este método também precisa chamar o método correspondente da classe-pai para fazer a mesma coisa.

Dessa maneira, o esboço deste método fica assim:

```
protected override void WndProc(ref Message m) {
    //
    // Processamento das mensagens personalizadas
    //
    base.WndProc(ref m); //equivalente a DefWindowProc
}
```

Definida como é comunicação entre processos, utilizando-se Windows Messages, na próxima seção iremos abordar como é a comunicação entre processos utilizando-se áreas de memória compartilhada.

5.5. MANIPULAÇÃO DE ÁREAS DE MEMÓRIA COMPARTILHADAS

Na seção anterior, vimos o que são as *Windows Messages* e como elas podem ser utilizadas para a comunicação entre dois programas Windows que possuam janelas.

No entanto, a comunicação entre processos usando *Windows Messages* não é muito eficiente quando se deseja transmitir muita informação de um processo a outro. Os parâmetros `wParam` e `lParam`, que transmitem informações adicionais da mensagens são apenas dois campos de 32 bits, sendo insuficientes para se transmitir uma *string*, por exemplo.

Para contornar esse problema, pode-se utilizar como mecanismo de comunicação entre processos as áreas de memória compartilhada. Assim, esta seção se propõe a explicar como se pode criá-las e como é possível manipulá-las.

Antes de começarmos a descrever o processo de manipulação, primeiro descreveremos o que são as áreas de memória compartilhada.

O Windows, como qualquer sistema operacional moderno, protege um programa, e os outros dele mesmo, definindo uma área de memória somente para o mesmo, quando este é criado.

Assim, se este programa em algum momento tentar acessar uma área de memória que não seja a definida para ele, o sistema operacional não o permitirá e lhe informará uma mensagem de erro famosa conhecida como falha de segmentação (*segmentation fault*), i.e., falha na tentativa de se acessar um segmento de memória que não é destinada ao mesmo.

E, da mesma maneira que este processo não consegue acessar a área de memória de outro, esse outro não consegue acessar a memória do primeiro, estando a memória de qualquer processo em *user mode* protegida de uma escrita acidental por outro processo[46].

[46] Conforme visto no capítulo 4, em *kernel mode* não existe esse mecanismo de proteção de memória de cada processo pelo sistema operacional Windows, o que pode causar muitos transtornos.

Às vezes, um processo precisa de mais memória que ele possui ou simplesmente deseja alocar uma memória que originalmente não pertence a ele, utilizando métodos de alocação dinâmica de memória como o famoso `malloc`.

Essa função simplesmente pede ao sistema operacional que este crie uma área de memória com o tamanho especificado e lhe permita acessá-la.

Da mesma maneira, é possível que um determinado programa peça ao sistema operacional que crie uma área de memória e lhe dê acesso e outro programa peça acesso a esta área de memória criada pelo primeiro. Assim, será possível que dois programas acessem a mesma área de memória e não seja causado falha de segmentação, porque ao sistema operacional foi pedido permissão de acesso pelos dois programas a esta área de memória, sendo a mesma concedida. Esta área de memória é chamada **área de memória compartilhada**.

A manipulação dessas áreas é realizada, basicamente, em duas etapas. Primeiro pede-se para esta área ser criada ou aberta, se ela já tiver sido criada. Depois, os programas envolvidos pedem para que esta área de memória seja mapeada, i.e., o sistema operacional forneça um ponteiro para a mesma.

A criação de áreas de memória compartilhadas é realizada utilizando-se a função `CreateFileMapping`[47], cujo cabeçalho é descrito abaixo:

```
HANDLE WINAPI CreateFileMapping(
  __in      HANDLE hFile,
  __in_opt  LPSECURITY_ATTRIBUTES lpAttributes,
  __in      DWORD flProtect,
  __in      DWORD dwMaximumSizeHigh,
  __in      DWORD dwMaximumSizeLow,
  __in_opt  LPCTSTR lpName
);
```

Os parâmetros dessa função são os definidos na tabela abaixo:

[47] http://msdn.microsoft.com/en-us/library/aa366537(VS.85).aspx

Parâmetro	Significado
hFile	*Handle* para o arquivo (na criação de áreas de memória, deve-se passar INVALID_HANDLE_VALUE)
lpAttributes	Estrutura de dados SECURITY_ATTRIBUTES[48], contendo parâmetros de segurança, como a possibilidade dessa área de memória ser utilizada pro processo filhos. Geralmente, coloca-se esse campo como NULL
flProtect	Parâmetro de proteção da área de memória, indicando se a mesma será usada para escrita, leitura ou mesmo execução
dwMaximumSizeHigh	Parte alta do valor especificado como tamanho da memória alocada
dwMaximumSizeLow	Parte baixa do valor especificado como tamanho da memória alocada
lpName	Nome da área de memória a ser criada

TAB. 5-5: Argumentos da função CREATEFILEMAPPING e seus respectivos significados

Essa função retorna um *handle* para a área de memória criada. Caso a função falhe, será retornado NULL.

Enquanto que um determinado processo criará a área de memória compartilhada com esta função, outro processo que deseja compartilhá-la a abrirá utilizando a função OpenFileMapping[49], descrita pelo cabeçalho abaixo:

```
HANDLE WINAPI OpenFileMapping(
  __in  DWORD dwDesiredAccess,
  __in  BOOL bInheritHandle,
  __in  LPCTSTR lpName
);
```

Esta função recebe três parâmetros como se pode ver, sendo o primeiro um inteiro que descreve o tipo de acesso que se deseja fazer na área de memória acessada (geralmente FILE_MAP_ALL_ACCESS); o segundo uma *flag* especificando se a área de memória a ser acessada pode ser acessada por processos filhos; e o terceiro especificando o nome da área de memória a ser aberta.

O retorno dessa função é um *handle* para a área de memória aberta. Em caso de falha, a função retorna NULL.

Deve-se ressaltar que o nome passado no argumento lpName de ambas as funções deve ser exatamente o mesmo para que a área de memória a ser passada seja a mesma.

[48] http://msdn.microsoft.com/en-us/library/aa379560(VS.85).aspx
[49] http://msdn.microsoft.com/en-us/library/aa366791(VS.85).aspx

Após ambos os processos envolvidos criarem ou abrirem a área de memória (podendo ser sem problemas mais de dois processos a acessar a área de memória em questão), eles devem mapear a área de memória, pedindo ao sistema operacional um ponteiro para a mesma. Isto é feito através da função `MapViewOfFile`[50], cujo cabeçalho é descrito abaixo:

```
LPVOID WINAPI MapViewOfFile(
  __in  HANDLE hFileMappingObject,
  __in  DWORD dwDesiredAccess,
  __in  DWORD dwFileOffsetHigh,
  __in  DWORD dwFileOffsetLow,
  __in  SIZE_T dwNumberOfBytesToMap
);
```

A descrição dos parâmetros é realizada na tabela abaixo:

Parâmetro	Significado
`hFileMappingObject`	*Handle* para a área de memória a ser mapeada (o *handle* recebido pelas funções anteriores de criação e abertura dessas áreas)
`dwDesiredAccess`	Campo de 32 bits especificando o tipo de acesso a ser realizado nessa área de memória a ser mapeada (usualmente se utiliza `FILE_MAP_ALL_ACCESS`)
`dwFileOffsetHigh`	Parte alta do valor especificado como deslocamento do endereço dentro da área de memória onde será o início da região a ser mapeada
`dwFileOffsetLow`	Parte baixa do valor especificado como deslocamento do endereço dentro da área de memória onde será o início da região a ser mapeada
`dwNumberOfBytesToMap`	Tamanho em *bytes* da área de memória a ser mapeada

TAB. 5-6: Argumentos da função MAPVIEWOFFILE e seus respectivos significados

Como se pode ver em seu cabeçalho, a função, em caso de sucesso, retorna um ponteiro `void *` apontando para o começo da área de memória compartilhada. Em caso de falha, a função retorna NULL.

O erro mais comum para se ter uma falha é o número de *bytes* a ser mapeado ser fornecido incorretamente.

Com esses ponteiros para o começo da área de memória compartilhada, ambos os processos que compartilham essa área de memória podem agora se comunicar, fornecendo

[50] http://msdn.microsoft.com/en-us/library/aa366761(VS.85).aspx

informações longas como *strings*, bastando que um processo escreva na área de memória um *array* de `char`, enquanto outro lê esse *array*.

Ao final da utilização da área de memória compartilhada, é importante que ela seja liberada, bem como os valores dos *handles* recebidos pelos processos sejam devolvidos ao sistema operacional.

Para fazer com que o mapeamento de uma área de memória compartilhada seja desfeito em um programa utiliza-se função `UnmapViewOfFile`[51], descrita pelo cabeçalho abaixo:

```
BOOL WINAPI UnmapViewOfFile(LPCVOID lpBaseAddress);
```

Tal função recebe como parâmetro o ponteiro para o início da área de memória compartilhada e retorna `TRUE`, em caso de sucesso, ou `FALSE`, em caso contrário. Ela desvincula o processo a área de memória compartilhada. Quando não restar mais nenhum processo vincluado a esta área de memória, o sistema operacional a destrói.

Após fazer isto, deve-se liberar o *handle* dessa área de memória para o sistema. Isto é feito utilizando-se a função `CloseHandle`[52], descrita abaixo:

```
BOOL WINAPI CloseHandle(HANDLE hObject);
```

Ela recebe como parâmetro o *handle* a ser devolvido para o sistema, retornando `TRUE`, em caso de sucesso, e `FALSE`, em caso negativo.

Deste modo, conforme visto, estes dois meios de comunicação de processos descritos nessas duas últimas seções são excelentes meios de comunicação entre processos e, conforme será mostrado no capitulo referente a arquitetura do sistema, foram utilizados no funcionamento do protótipo desenvolvido.

5.6. *DYNAMIC-LINK LIBRARIES* (DLLs)

O Windows fornece um interessante e poderoso recurso que se chama "**linkagem" dinâmica de bibliotecas**, que permite que programas utilizem em tempo de execução procedimentos, funções e até recursos como imagens de uma biblioteca disposta em forma de arquivo na memória secundária (em disco).

Originalmente esse recurso não foi criado no Windows, mas sim em um sistema operacional muito antigo denominado MULTICS, que foi o precursor do UNIX, no final da década de 60.

---

[51] http://msdn.microsoft.com/en-us/library/aa366882(VS.85).aspx
[52] http://msdn.microsoft.com/en-us/library/ms724211(VS.85).aspx

Esse recurso foi adotado pelos desenvolvedores do Windows ainda em suas primeiras versões, quando não havia divisão do espaço de memória principal e precisava-se eliminar a redundância em programas que realizavam o mesmo conjunto de instruções.

Através dessas bibliotecas, ao invés de dois programas conterem o código de uma determinada função para ser executados, eles poderiam se "linkar" com uma determinada biblioteca e utilizar essa determinada função em comum disponibilizada pela mesma. Isso eliminava a redundância na medida em que o código em comum não precisava ser escrito duas vezes.

Isso era muito importante principalmente para as funções da API do Windows, que antes da adoção desta técnica, eram dispostas na forma de programas executáveis que eram chamados pelos programas comuns.

A eliminação da redundância de código não é a única funcionalidade existente nas DLLs. O mais interessante dessas bibliotecas é a possibilidade de se poder utilizá-las em tempo de execução do programa.

Esta seção descreverá como isto pode ser feito, não sendo de maneira nenhuma abordado como se constrói essas bibliotecas, que, diga-se de passagem, são independentes da linguagem em que se escreve.

Para realizar a "linkagem" dinâmica (em tempo de execução de programa) de uma biblioteca DLL, utiliza-se a função `LoadLibrary`[53], cujo cabeçalho é descrito abaixo:

```
HMODULE WINAPI LoadLibrary(LPCTSTR lpFileName);
```

Tal função recebe como parâmetro uma *string* contendo o caminho completo da localização da DLL no sistema operacional e retorna um *handle* para a biblioteca "linkada".

Esta função carrega a biblioteca, alocando-a no espaço de memória do processo que a chama. Assim, se dois processos carregam a mesma biblioteca, essa não é alocada apenas uma vez, mas uma vez para cada processo que a requisita.

Uma vez carregada esta função, utiliza-se a função `GetProcAddress`[54] para se obter o endereço de uma função disponibilizada por esta biblioteca para uso externo. Seu cabeçalho é descrito abaixo:

```
FARPROC WINAPI GetProcAddress(
  __in  HMODULE hModule,
  __in  LPCSTR lpProcName
);
```

[53] http://msdn.microsoft.com/en-us/library/ms684175(VS.85).aspx
[54] http://msdn.microsoft.com/en-us/library/ms683212(VS.85).aspx

Como se vê, esta função recebe como parâmetros o *handle* da biblioteca “linkada” anteriormente e uma *string* contendo o nome da função a ser carregada.

O retorno dessa função é um ponteiro de função para a função desejada. Através desse ponteiro é que se pode chamar uma função disponibilizada por essa biblioteca.

Esse procedimento descrito nesta seção é muito importante para o desenvolvimento do protótipo, uma vez que a LibClamAv é disposta na forma de DLL.

5.7. CRIAÇÃO DE *THREADS*

Nos primórdios da computação, os sistemas operacionais eram desenvolvidos para serem *single-tasked*, i.e., apenas um programa por vez era executado pela CPU da máquina que continha o sistema operacional. Esses programas desenvolvidos para esses sistemas operacionais possuíam ao final de seus códigos uma instrução *yield* que devolvia o processamento da CPU o sistema operacional.

Para melhorar esta limitação, foram criados os sistemas operacionais *multi-tasked* (sendo o Windows um deles), em que cada programa possui um determinado tempo, denominado *time slice*, para ser executado na CPU. Ao final desse tempo, se o programa não terminasse sua tarefa, era então interrompido e somente restaurado até que todos os outros programas passassem pela CPU.

Paralelamente ao conceito de sistema operacional *muti-tasked*, foi-se criado o conceito de *threads*.

Às vezes, era desejoso que um programa, ao invés de dispender todo o seu *time slice* em uma única seqüência de código, dispendesse seu tempo em mais de uma seqüência de instruções, de forma a simular uma paralelização de suas atividades.

Cada uma dessas seqüências de códigos, que é executada dentro do tempo destinado a um processo, é denominada *thread*. São como mini-processos que são executados de forma independente uns dos outros.

A única grande diferença das *threads* para os processos é que as *threads* não possuem proteção de memória em relação a área de memória pertencente ao mesmo processo. Isto é obviamente uma vantagem, quando deseja-se paralelizar uma atividade, mas não se deseja criar processos filhos.

Esta seção pretende descrever como se faz para criar um program *muli-thread*.

Para o sistema operacional, todo o processo que se inicia possui uma *thread* inicial denominada ***Main Thread***.

Pode-se ao longo da execução da mesma, criar outras *threads* adicionais. Isto é feito utilizando-se a função `CreateThread`[55], cujo o cabeçalho é descrito abaixo:

```
HANDLE WINAPI CreateThread(
  __in_opt   LPSECURITY_ATTRIBUTES lpThreadAttributes,
  __in       SIZE_T dwStackSize,
  __in       LPTHREAD_START_ROUTINE lpStartAddress,
  __in_opt   LPVOID lpParameter,
  __in       DWORD dwCreationFlags,
  __out_opt  LPDWORD lpThreadId
);
```

Essa função possui os seguintes parâmetros descritos pela tabela abaixo:

Parâmetro	Significado
`lpThreadAttributes`	Ponteiro para a estrutura `SECURITY_ATTRIBUTES`, que determinará se o *handle* retornado por esta função pode ser herdado para processos filhos
`dwStackSize`	Tamanho da pilha destinada a esta *thread* (se for zero, o tamaho da pilha será o padrão para *threads*)
`lpStartAddress`	Ponteiro para a função principal da *thread*
`lpParameter`	Parâmetros para a função principal da *thread* (se não houver parâmetros na função de entrada, deve-se colocar `NULL`)
`dwCreationFlags`	Parâmetros de criação da *thread*, indicando se ela começa suspensa ou não
`lpThreadId`	Ponteiro para um `DWORD` que receberá o valor de retorno da função principal (caso a função seja do tipo `VOID`, deve-se colocar `NULL`)

TAB. 5-7: Argumentos da função CREATETHREAD e seus respectivos significados

O retorno dessa função é um *handle* para a *thread* criada, que pode ser utilizado para por exemplo suspender seu estado.

5.8. ACESSO AO REGISTRO DO WINDOWS

Um dos componentes mais importantes do Windows é o **Registro do Windows**, que começou a existir no Windows 95.

[55] http://msdn.microsoft.com/en-us/library/ms682453(VS.85).aspx

Trata-se de um banco de dados que centraliza quase todas as informações de configurações do sistema, desde coisas fúteis como a cor do fundo e das letras do prompt do DOS[56], passando pro informações sigilosas como as senhas criptografadas dos usuários[57] e indo até informações vitais ao sistema como a ordem dos grupos de *drivers* que serão carregados durante a inicialização do Windows[58].

A sua organização se faz através de chaves que são estruturadas da mesma maneira que os diretórios. Cada chave pode estar contida dentro de outra chave ou de uma chave-raiz, denomindada HKEY. Cada chave também contém um conjunto de valores, que são acessados segundo um *hash*, em que através do nome da chave, obtém-se um dado armazenado.

São inúmeras as aplicações que se valem do Registro do Windows para armazenarem informações relevantes. O Registro do Windows é um lugar excelente para armazenar alguns parâmetros de configuração por ser de fácil acesso e independente do local de instalação de um software, sem contar com o fato de que outros programas podem utilizar determinadas informações do Registro para interagirem com seu software, como é o caso do Painel de Controle do Windows que pode catalogá-lo caso determinados valores sejam adicionados em uma determinada chave do Windows

Esta seção trata de como se realiza o acesso as chaves do registro do Windows e seus valores contidos. Em C, o acesso ao Registro do Windows é basicamente realizado por três funções, `RegOpenKeyEx`, `RegQueryValueEx` e `RegCloseKey`, disponíveis `Winreg.h` em e implementado pela biblioteca `Advapi32.dll`. Os cabeçalhos dessas funções são descritos abaixo:

```
LONG WINAPI RegOpenKey(HKEY hKey, LPCTSTR lpSubKey, PHKEY phkResult);
LONG WINAPI RegQueryValueEx(HKEY hKey, LPCTSTR lpValueName, LPDWORD lpReserved, LPDWORD lpType, LPBYTE lpData, LPDWORD lpcbData);
LONG WINAPI RegCloseKey(HKEY hKey);
```

A primeira função, `RegOpenKey`, abre uma determinada chave do registro. Ela recebe como argumento respectivamente um *handle* para uma chave já aberta ou para alguma chave-

[56] Armazenada no valor `DefaultColor` da chave `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Command Processor`.

[57] Embora normalmente não se possa acessar seu conteúdo mesmo como administrador, as senhas ficam gravadas em `HKEY_LOCAL_MACHINE\SECURITY\SAM\Domains\Account\Users` nas sub-chaves correspondentes a cada usuário.

[58] Armazenadas no valor `List` da chave `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder`

raiz, um ponteiro para uma *string* contendo a chave que se deseja abrir e um ponteiro para um *handle* que armazenará o identificador da chave que será aberta.

A segunda função, `RegQueryValueEx`, é a responsável por obter um dado de um valor de uma chave já aberta. A tabela abaixo explicita os significados de seus parâmetros:

Parâmetro	Significado
`hKey`	*Handle* para a chave aberta
`lpValueName`	Nome do valor da chave, cujo dado deseja-se obter
`lpReserved`	Reservado (deve ser preenchido como `NULL`)
`lpType`	Tipo de dado a ser pesquisado
`lpData`	*Buffer* para armazenamento do dado
`lpcbData`	Ponteiro para um inteiro que armazenará o tamanho do dado armazenado

TAB. 5-8: Argumentos da função REGQUERYVALUE e seus respectivos significados

Já a última função, `RegCloseKey`, é responsável por fechar a chave do registro. Recebe obviamente como parâmetro o *handle* para a chave a ser fechada.

5.9. PESQUISA DE ARQUIVOS EM SUBDIRETÓRIOS

Muitas vezes se faz necessário descobrir o nome dos arquivos e subdiretórios contidos num determinado diretório. Esta seção trata de como esse procedimento, denominado pesquisa de arquivos em diretórios, pode ser realizada na linguagem C.

A API do Windows fornece funções disponíveis no arquivo `WinBase.h` e implementadas na biblioteca `Kernel32.dll` para tal funcionalidade. Basicamente, realiza-se a pesquisa utilizando-se três funções: `FindFisrtFile`[59], `FindNextFile`[60] e `FindClose`[61].

Primeiro, declara-se uma estrutura de dados `WIN32_FIND_DATA`[62]. Esta estrutura armazenará os dados do resultado da pesquisa. Basicamente, dois campos dela são utilizados: o campo `cFileName`, que armazena o nome do arquivo pesquisado, e o campo `dwFileAttributes`, que armazena os atributos desse arquivo, necessário para descobrir se este arquivo é um diretório.

[59] http://msdn.microsoft.com/en-us/library/aa364418(VS.85).aspx
[60] http://msdn.microsoft.com/en-us/library/aa364428(VS.85).aspx
[61] http://msdn.microsoft.com/en-us/library/aa364413(VS.85).aspx
[62] http://msdn.microsoft.com/en-us/library/aa365740(VS.85).aspx

O primeiro arquivo a ser pesquisado deve ser feito por intermédio da função , cujo cabeçalho é o descrito abaixo:

```
HANDLE FindFirstFile(LPCTSTR lpFileName, LPWIN32_FIND_DATA lpFindData);
```

Tal função recebe como argumentos respectivamente um ponteiro para uma string contendo o nome do diretório a ser pesquisado e o endereço da estrutura de dados que armazenará o resultado da pesquisa, contendo o nome do primeiro arquivo pesquisado. O retorno da função é um *handle*, que contém um identificador único para a pesquisa.

É importante considerar que para a pesquisa ser bem-sucedida, a string do diretório a ser pesquisado deve ser precedido de "*", i.e., se deseja-se pesquisar no diretório "C:\", essa string deverá ser "C:*".

Caso a pesquisa seja mal-sucedida e não seja possível iniciar a pesquisa o valor retornado pela função será `INVALID_HANDLE_VALUE`. A função `GetLastError()`[63] pode ser usada para especificar a causa deste erro, sendo provavelmente o arquivo não ser encontrado.

Os arquivos subseqüentes deverão ser pesquisados pela função `FindNextFile`.

```
BOOL FindNextFile(HANDLE hFindFile, LPWIN32_FIND_DATA lpFindData);
```

Esta função recebe como parâmetros respectivamente o *handle* retornado pela função `FindFistFile` e o endereço da estrutura de dados que armazena o resultado da pesquisa. O retorno dessa função é 1 se a pesquisa é bem sucedida e 0, caso esta não seja. É interessante notar que caso não haja mais arquivos a serem pesquisado essa função retorna 0 e a função `GetLastError()` retorna como identificador de erro `ERROR_NO_MORE_FILES`.

Outra informação digna de nota é que a pesquisa geralmente retorna respectivamente como os dois primeiros arquivos "." e "..", que indicam o nível corrente do diretório e o nível acima. Na maioria das vezes, essas duas informações são inúteis e devem ser desconsiderada na pesquisa.

Para encerrar a pesquisa é utilizada a função `FindClose`, que recebe como parâmetro o *handle* da pesquisa.

```
BOOL FindClose(HANDLE hFindFile);
```

Assim, de maneira geral a pesquisa é realizada da seguinte forma:

---

[63] `DWORD GetLastError(void);`
(para mais detalhes, consulte: http://msdn.microsoft.com/en-us/library/ms679360(VS.85).aspx)

```
WIN32_FIND_DATA findData;
   HANDLE h;
   (...)
   h = FindFirstFile(folderpath, &findData);
   // nome do arquivo contido em findData.cFileName
   if(h != INVALID_HANDLE_VALUE) {
     (...)
     while(FindNextFileA(h, &findData)) {
     // nome do arquivo contido em findData.cFileName
     (...)
     }
   }
   FindClose(h);
```

6. NULLSOFT SCRIPTABLE INSTALL SYSTEM (NSIS)

6.1. INTRODUÇÃO

Nullsoft Scritable Install System (NSIS) é um sistema de instalação, criado pela Nullsoft, criadora do famoso Winamp, que permite a construção de um instalador para Windows através de uma linguagem de *script*.

Esse sistema foi criado originalmente pela necessidade de se distribuir o Winamp. A primeira versão do NSIS, a 1.0, era bem básica contendo apenas alguns comandos para a criação de diretórios e para a criação de chaves de registro do Windows.

Mais tarde, após a versão 2.0a, ele foi transferido para a SourceForge e outros desenvolvedores o melhoraram, permitindo, por exemplo, a adição de interfaces gráficas. Atualmente ele é utilizado pro diversos programas livres devido a simplicidade e facilidade de sua linguagem de *script*.

A confecção do instalador é realizada da seguinte maneira pelo sistema: o NSIS compila o *script* escrito, comprime os arquivos que serão "deployados" na instalação (geralmente através do BZIP) e cria um executável contendo o resultado dessa compressão e das instruções compiladas.

Esse capítulo tem por finalidade descrever um pouco dessa linguagem para a confecção de um instalador simples, descrevendo os principais comandos que este script deve conter.

6.2. PRINCIPAIS CONSTANTES

Todas as variáveis do script são precedidas pelo $.

A tabela a seguir descreve as principais variáveis constantes que são úteis na confecção do instalador:

Constante	Significado
`$INSTDIR`	Diretório de instalação
`$SMPROGRAMS`	Diretório onde fica localizada a pasta “Programas” do menu iniciar
`$DOCUMENTS`	Diretório onde fica localizada a pasta “Meus Documentos”
`$DESKTOP`	Diretório onde ficam localizados os itens do *Desktop*
`$PROGRAMFILES`	Localização da pasta “Arquivos de Programas”
`$OUTDIR`	Atual diretório de saída que alterado pro vários comandos

TAB. 6-1: Principais constantes do NSIS

6.3. PRINCIPAIS COMANDOS GLOBAIS

Os principais comandos globais existentes em NSIS são: `Name`, `OutFile`, `RequestExecutionLevel`, `SilentInstall` e `InstallDir`.

O primeiro define o nome do instalador que aparecerá na interface gráfica.

O segundo define o nome do arquivo executável do instalador.

O terceiro, somente válido para o Windows Vista, define o tipo de permissão que se deve ter para se rodar este instalador.

O quarto define se o instalador é silencioso, i.e., não possui nenhuma interação com o usuário.

O último define o diretório padrão de instalação.

6.4. FUNÇÕES

Pode se definir funções na linguagem de script do NSIS. As funções em NSIS são na verdade procedimentos, visto que não possuem valor de retorno.

Elas são definidas utilizando-se as palavras reservadas `Function` e `EndFunction`, que definem o escopo do código dessas funções.

O nome da função deve ser declarado, logo após a palavra reservada `Function`. Algumas funções especiais podem ser sobrescritas. Essas funções possuem seus nomes precedidos por “`.`” ou “`un.`” .

6.5. PÁGINAS

A menos que o instalador seja definido como silencioso, este interage com o usuário em etapas, que são definidas pelas páginas.

O NSIS permite ao desenvolvedor do instalador definir quais dessas etapas farão parte do processo de instalação. Isto é feito utilizando-se os comandos `Page`, `PageEx` e `PageExEnd`.

O comando Page é utilizado da seguinte forma:

```
Page tipo_de_pagina
```

Onde `tipo_de_pagina` é um tipo já preestabelecido de página de instalação, podendo ser:

- `license`: etapa em que se mostra o usuário um texto sobre a licença de uso e pedindo que o usuário concorde com o texto para dar procedimento a instalação;

- `components`: etapa em que o usuário determina os componentes que serão instalados;

- `directory`: etapa em que se pergunta ao usuário o local de instalação dos arquivos;

- `instfile`: etapa em que se mostra ao usuário uma tela, mostrando o progresso de instalação;

- `custom`: etapa adicional onde uma determinada função é executada. O nome dessa função deve ser mencionado logo após a palavra `custom`;

É possível personalizar certos elementos dessas páginas, o que é feito através dos comandos `PageEx` e `PageExEnd`. Por exemplo, pode-se definir os rótulos da tela onde será perguntado o diretório de instalação da seguinte maneira:

```
PageEx directory
DirText [texto] [sub-texto] [texto do botão browse] [texto da caixa de diálogo do browse]
PageExEnd
```

6.6. SEÇÕES

Os principais comandos de instalação são definidos no escopo definido entre os comandos `Section` e `SectionEnd`. São eles que definem a seqüência de comandos que farão as coisas mais importantes como o *deploy* dos arquivos a serem instalados.

Os principais comandos utilizados são: `SetOutPath`, `File` e `WriteRegStr`.

O primeiro serve para a criação ou a abertura de um diretório. É usado da seguinte maneira:

```
SetOutPath diretório
```

Se esse diretório existir, ele é somente carregado. Caso contrário, não só ele como todos os subseqüentes são criados.

O segundo serve para copiar um arquivo contido na compressão do isntalador para o diretório corrente. É usado da seguinte maneira:

```
File arquivo
```

Onde arquivo é o nome do arquivo contido na compressão do instalador.

Por último, temos um comando que serve para a criação de valores em chaves do Registro do Windows. É utilizado da seguinte forma:

```
WriteRegStr chave-raiz sub-chave nome-do-valor dado
```

Onde seus argumentos são respectivamente a chave-raiz (HKEY) onde se deseja escrever esse valor, a sub-chave onde está esse valor, o nome do valor a ser procurado no registro e o dado a ser escrito.

Se, ou a chave, ou o valor não existirem no registro, esses serão criados antes da escrita.

7. ARQUITETURA E FUNCIONAMENTO DO SISTEMA

7.1. ARQUITETURA PROPOSTA

Após a explicação dos diversos conceitos básicos necessários à compreensão do protótipo, como a visão geral de um anti-vírus, a compreensão do ClamAv, dos *drivers* do Windows, de diversas partes da API do Windows e do que é o NSIS, podemos agora entrar mais a fundo nos detalhes do desenvolvimento do protótipo.

Assim, a seguir delinearemos a arquitetura escolhida para o protótipo de anti-vírus, sua subdivisão em diversos componentes, o funcionamento do protótipo como um todo e o funcionamento de cada um dos componentes.

Durante o desenvolvimento do protótipo de anti-vírus estático, este foi batizado com o nome de **ClamEB**, proveniente da junção das siglas ClamAv com EB (Exército Brasileiro).

Especificamente nessa seção, será abordada a arquitetura de sistema proposta para o ClamEB, sendo explicitado como este é dividido em componentes.

Conforme foi dito anteriormente, a arquitetura do ClamEB foi profundamente influenciada pela arquitetura do ClamWin. Basicamente, o protótipo é dividido em cinco componentes. A figura abaixo traz uma idéia de como é a divisão do ClamEB em componentes e como é a relação de dependências entre os componentes:

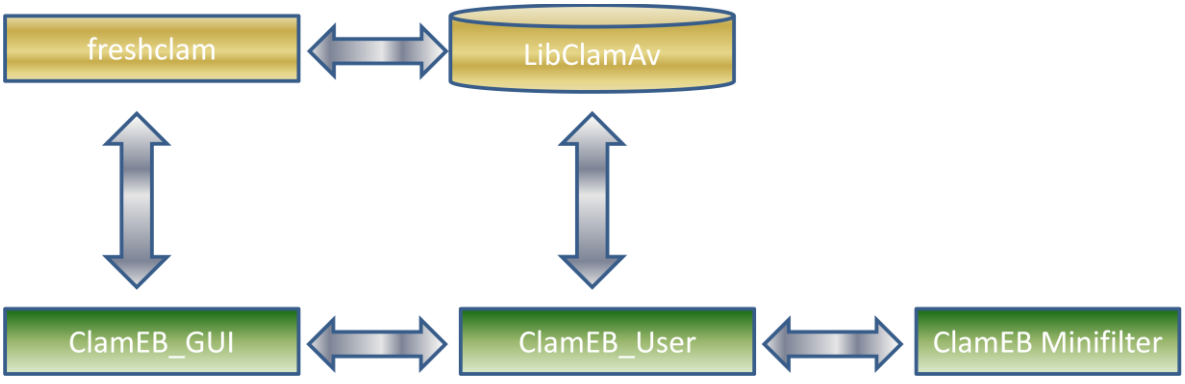

FIG. 7-1: Componentes do ClamEB

Podemos notar que dois dos cincos componentes são próprios do ClamAv e não foram implementados nesse projeto, sendo obtidos já prontos. São eles: a LibClamAv e o freshclam.

A **LibClamAv** é o componente responsável por toda a *engine* do ClamAv que faz a verificação de assinaturas virais nos arquivos e lida com a base de dados do ClamAv dipostas

nos arquivos de formato CVD. A LibClamAv é composta das bibliotecas `libclamav.dll` e `pthreadvc2.dll`.

O **freshclam** é o componente responsável pela conexão do programa com os *mirrors* do ClamAv e com o processo em si de atualização da base de dados. É importante notar que ele é dependente das bibliotecas da LibClamAv.

O **ClamEB Minifilter** é o componente responsável pela interceptação das requisições de acesso ao *file system* do sistema operacional Windows. É ele que verifica que um determinado arquivo está sendo acessado e notifica o ClamEB_User sobre o nome do arquivo.

O **ClamEB_User** é o componente que desempenha o papel de *middleware* no sistema. É ele que faz o interfaceamento entre o *minifilter*, a LibClamAv e a interface gráfica. É ele que recebe as notificações do *minifilter*, utiliza as funções da LibClamAv para escanear os arquivos e notifica a interface gráfica em caso de contaminação viral.

O **ClamEB_GUI** é o componente que faz o papel da GUI (*Graphical User Interface*). Ele é responsável por toda a interação com o usuário através de uma interface gráfica.

Devemos destacar que além desses cinco componentes, o ClamEB possui um sexto componente que é o instalador.

A seguir, serão apresentadas as tecnologias utilizadas na construção desses três componentes que foram desenvolvidos, para mais tarde explicá-los um a um mais profundamente.

7.2. TECNOLOGIAS UTILIZADAS

Três linguagens de programação (C, C#, NSIS) foram utilizadas na construção desses componentes e várias ferramentas foram empregadas.

O instalador do ClamEB foi feito utilizando a linguagem de script NSIS, que já fora apresentada anteriormente. A interface gráfica ClamEB_GUI foi construída utilizando C# (.NET). O ClamEB_User e o ClamEB Mini Filter foi construído utilizando uma linguagem C estendida da Microsoft.

É importante destacar que, como a interface gráfica foi construída usando C#, a plataforma .NET 2.0 deve estar instalada no sistema operacional Windows em que se deseja executar o protótipo de anti-vírus[64].

[64] Na maioria das versões do Windows Vista de hoje em dia, a plataforma .NET já vem instalada por padrão, o que não ocorre nos outros Windows como o XP. Para instalar a plataforma .NET nesses sistemas operacionais, siga as instruções dispostas em:

Para desenvolvimento da interface gráfica foi utilizado o *Microsoft Visual C# 2005 Express Edition*. Essa ferramenta permite a construção de interfaces gráficas em C#, bem como a edição de código e a compilação do programa. Ela é uma ferramente gratuita, sendo de uso ilimitado desde que se registre como usuário da mesma.

Para escrita do código do ClamEB_User e ClamEB Mini Filter foi utilizado o *Microsoft Visual C++ 2005 Express Edition* que, assim como *Visual C# 2005*, tambem é gratuito e ilimitado desde que seja registrado.

Para compilação do ClamEB_User e do ClamEB Mini Filter foi utilizado o *Windows Driver Kit* (WDK). O WDK provê vários ambientes de *build* (*build environments*), com uma ferramenta semelhante à ferramenta *make*, nos quais o desenvolvedor pode gerar os binários para as versões mais atuais do Windows (2000, XP, Server 2003, Vista e Server 2008), para as principais arquiteturas do mercado (x86, x64 e IA-64). No caso específico do presente trabalho, foi utilizado o *build environment* do Windows XP voltado para a arquitetura x86.

Para a realização do testes funcionais e de integração do sistema, devido ao risco de danificar o sistema com um *driver* ainda em desenvolvimento e não totalmente confiável, foi utilizada uma máquina virtual do programa *Microsoft Virtual PC 6.0.156.0*, um *software* criado pela Microsoft para concorrer com outros, como *VMWare*, no mercado de *softwares* de virtualização. Ele permite virtualizar os recursos fornecidos pelo *hardware* subjacente, permitindo executar simultaneamente dois ou mais sistemas operacionais em uma mesma máquina. No caso específico do presente trabalho, foi utilizada uma imagem de um sistema operacional Windows XP, provida pela própria Microsoft.

Para não utilizarmos arquivos infectados reais, que no seu manuseio poderiam prejudicar e danificar o sistema, utilizamos o arquivo de teste do *EICAR test*, descrito mais em detalhes em um apêndice na seção 11.2.

7.3. CLAMEB MINIFILTER

O componente ClamEB Minifilter, conforme foi visto, é um *file system filter driver*, ou mais especificamente falando, um *minifilter*. Pelo fato de ser um *kernel mode* driver, ele teve de ser criado em linguagem C e compilado e construído com o *build environment* do WDK.

Para a implementação da funcionalidade de *on-access scanning*, devemos filtrar as requisições de abertura de arquivos, verificando, antes de liberar o acesso a eles, se eles não

---

http://www.microsoft.com/downloads/details.aspx?familyid=0856eacb-4362-4b0d-8edd-aab15c5e04f5&displaylang=en

estão infectados. Portanto, investigando as descrições das principais operações de *file system* explicadas na TAB. 4-2, percebemos que estamos interessados nas requisições com IRP *major function code* igual a IRP_MJ_CREATE.

Para tratar essas requisições de abertura de arquivos, o CLAMEB MINIFILTER, cadastra 2 *callbacks* junto ao *Filter Manager*: uma *preoperation callback*, chamada `ClamEBPreCreate`, e uma *postoperation callback*, chamada `ClamEBPostCreate`.

A *postoperation callback* é necessária porque na *preoperation callback* desse tipo de operação não é possível obter qualquer informação do arquivo que se deseja abrir, simplesmente porque o *handle* daquele arquivo ainda não foi criado. No interstício entre a *preoperation* e a *postoperation callbacks*, o *handle* do arquivo é criado e torna-se possível acessar na *postoperation callback* os dados daquele arquivo.

Assim sendo, na função `ClamEBPostCreate`, as informações do arquivo são recuperadas, através do uso das funções adequadas da API de *minifilters*, o caminho (*path*) completo do arquivo é calculado e enviado, através do uso de *communication ports*, para o componente de *middleware*, o ClamEB_User, que será o responsável pela inspeção do arquivo para detectar se ele está infectado ou não.

Para evitar chamadas cíclicas e infinitas do sistema ao *minifilter*, causadas pelos acessos a arquivos do sistema por parte do ClamEB_User durante o funcionamento do anti-vírus, este componente deve, ao iniciar sua operação, se registrar junto ao *minifilter*. Este passará então a ignorar todas as chamadas de acesso ao *file system* vindas do ClamEB_User, pois este é considerado um componente confiável do sistema.

Como o minifilter intercepta todas as chamadas de acesso ao *file system*, uma inspeção de cada um deles em cada acesso, inclusive dos acessados pelo próprio sistema operacional, levaria muito tempo e consumiria uma quantidade razoável de ciclos do processador. Para evitar esse *overhead* grande, decidiu-se por filtrar no presente protótipo apenas as requisições referentes a arquivos com extensão EXE, COM, CMD e BAT, que são extensões clássicas de arquivos executáveis dos sistemas Windows.

7.4. CLAMEB_USER

Conforme citado anteriormente, o ClamEB_User é o componente do ClamEB que serve como *middleware*. Em outras palavras, este é o componente integrador do sistema, ligando-se a praticamente todos os componentes e realizando a interface com cada um deles.

Quando se realiza, por exemplo, o escaneamento manual de determinados arquivos, a interface gráfica recebe o comando do usuário para se escanear uma determinada pasta e informa ao ClamEB_User a localização dessa pasta. O ClamEB_User realiza a pesquisa de arquivos naquela pasta e, para cada arquivo, invoca a função da LibClamAv responsável pelo escaneamento. Cada arquivo infectado é informado, juntamente com o nome do vírus infectante e outras informações adicionais, é passado de volta para a interface gráfica que disponibliza essas informações ao usuário.

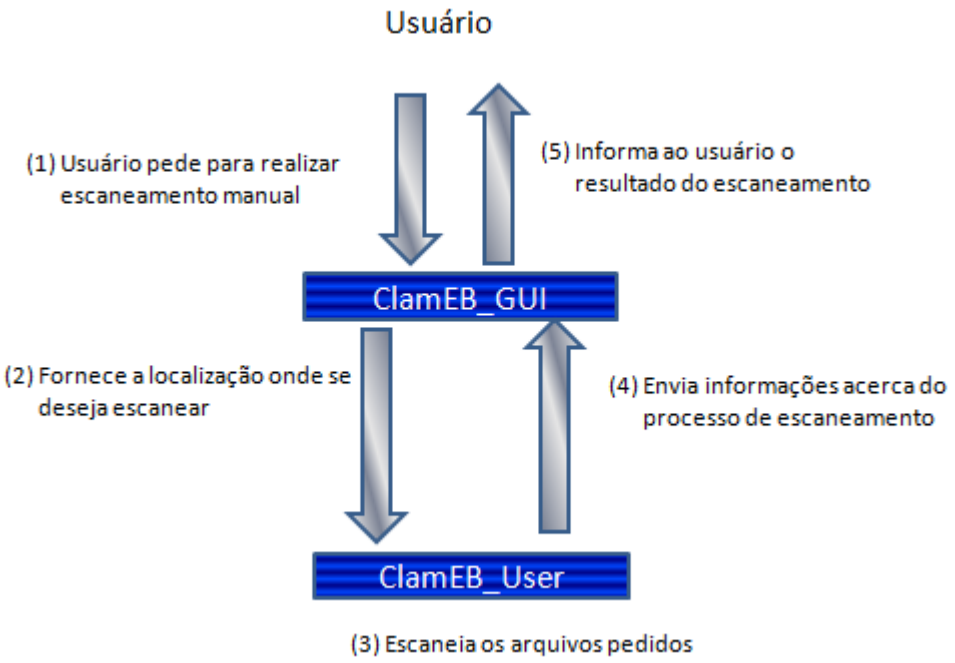

FIG. 7-2: Esquema do funcionamento do escaneamento manual

Quando acontece um escaneamento de um arquivo em tempo de acesso, o ClamEB Minifilter informa ao ClamEB_User o nome do arquivo a ser acessado. O ClamEB_User invoca a função da LibClamAv reponsável pelo escaneamento. Caso o arquivo esteja limpo, o ClamEB_User informa ao ClamEB Minifilter para liberar o acesso ao arquivo. Caso contrário, o ClamEB_User informa a interface gráfica o nome do arquivo infectado, bem como o nome do vírus que o infectou. A interface gráfica então expõe ao usuário uma mensagem para este decidir se deve liberar ou bloquear o acesso ao usuário e passa para o ClamEB_User a conclusão dessa decisão. Este, por sua vez, informa ao ClamEB Minifilter se deve bloquear ou não o arquivo, de acordo com a decisão do usuário.

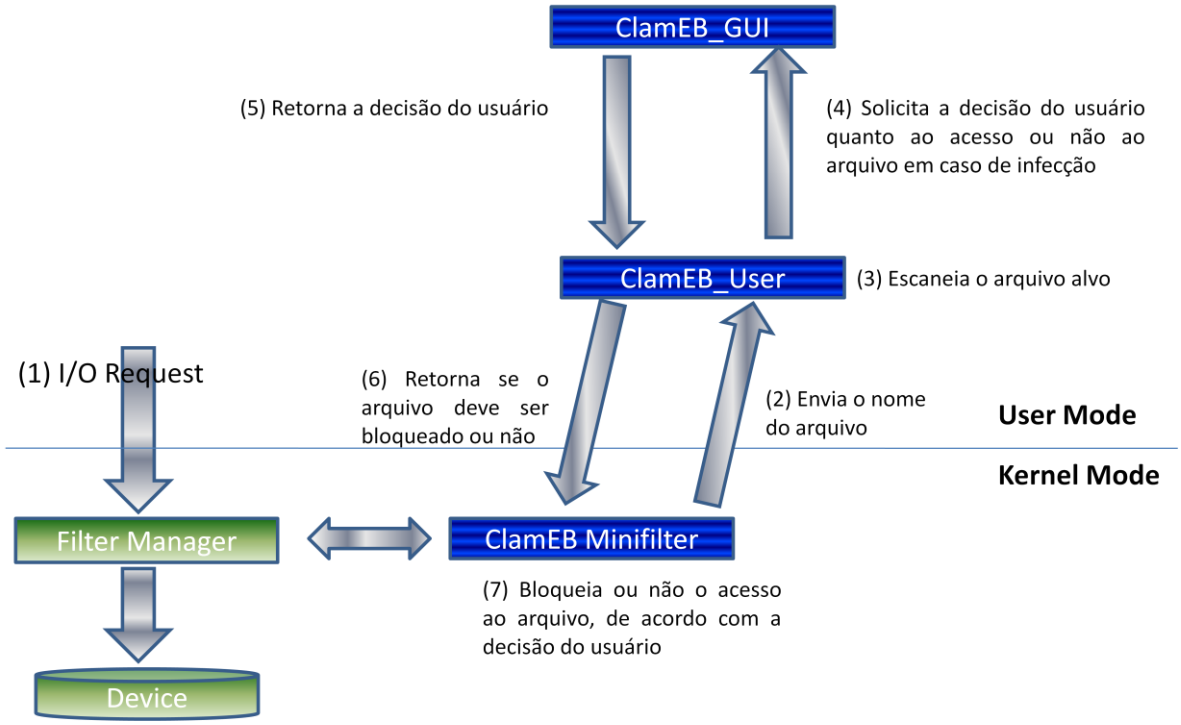

FIG. 7-3: Esquema do funcionamento do escaneamento *on-access*

Para que o ClamEB contemple todas as requisições de acesso a arquivos existentes, fez-se necessário que ele fosse desenvolvido para ser uma aplicação Windows *multi-thread*. Aplicação Windows, porque é necessária a criação de uma janela para que se possa realizar a comunicação com a interface gráfica utilizando-se *Window Messages*. E aplicação *multi-thread*, porque pode haver mais de uma requisição de acesso a arquivos ao mesmo tempo, sendo necessário tratar essas requisições no mesmo momento.

Assim, o ClamEB_User possui basicamente quatro tipos de *threads* em seu funcionamento:

Primeiro, tem-se a ***Main Thread***, que nada mais é que a *thread* principal da aplicação. Ela que realiza as etapas iniciais do serviço, que serão explicadas mais a frente, e que se responsabiliza pelo tratamento das *Window Messages*, recebidas pelo programa.

Em segundo lugar, tem-se a ***Manual Scan Thread***. Esta é a thread que sempre é iniciada quando o escaneamento manual de arquivos ou pastas é solicitado para ser realizado pela interface gráfica. É essa *thread* que realiza a pesquisa de arquivos na pasta desejada, invoca a função de escanemento da LibClamAv para cada arquivo pesquisado e repassa os resultados parciais de volta à interface gráfica.

Em seguida, temos a ***Filter Listening Thread***, iniciada logo após o estabelecimento da comunicação com a interface gráfica. Destina-se a receber todas as mensagens do ClamEB

Minifilter sobre arquivos que estão sendo acessados e, para cada um deles, cria uma *Worker Thread* para o seu tratamento.

Por último, temos as ***Worker Threads***, que, conforme dito anteriormente, fazem o tratamento dos arquivos cujo acesso é requisitado. Ela invoca a função de escaneamento da LibClamAv para este arquivo e verifica se o mesmo está infectado. Em caso negativo, informa ao ClamEB Minifilter para liberar o acesso e termina. Em caso positivo, informa a interface gráfica o nome do arquivo infectado e seu vírus. Em seguida, "dorme", só "acordando" quando a interface gráfica lhe informa se o usuário liberou ou não o acesso a este arquivo. Repassa esta decisão ao ClamEB Minifilter e termina.

Antes de o ClamEB_User entrar em perfeito funcionamento, ele passa por um estado inicial instável, onde algumas operações são realizadas. Essas operações são:

(1) Carregamento das configurações existentes no Registro do Windows (serão explicadas melhor mais tarde)
(2) Registro da WM_CLAMEB (*Window Message* personalizada)
(3) Registro e declaração da janela principal do ClamEB_user, que nunca será exibida
(4) Estabelecimento da comunicação com o ClamEB_Minifilter
(5) "Linkagem" da `libclamav.dll` e `pthreadvc2.dll`
(6) Criação da *engine* de escaneamento
(7) Estabelecimento da comunicação com a interface gráfica
(8) Criação da *Filter Listening Thread*

De modo a permitir todos esses detalhes de funcionamento, o ClamEB_User foi desenvolvido em módulos. A tabela abaixo descreve melhor como é a organização desses módulos do ClamEB_User, relacionando cada par de arquivos (*header* e C) com seu respectivo propósito.

Header	Arquivo C	Propósito
clamEB_User.h	clamEB_User.c	Procedimentos adotados na *Main Thread*
clamEB_UK.h	-	Estruturas comuns ao ClamEB_User e ClamEB Minifilter
globalvars.h	-	Estruturas de uso geral
clamav.h	-	Interface com a LibClamAv
settings.h	settings.c	Manipulação do Registro do Windows
scanengine.h	scanengine.c	Escaneamento de arquivos
shmem.h	shmem.c	Manipulação das áreas de memória compartilhadas
ipc.h	ipc.c	Manipulação das *Window Messages*
manualscan.h	manualscan.c	*Manual Scan Thread*
fltworker.h	fltworker.c	*Filter Listaening Thread* e *Worker Thread*

TAB. 7-1: Modularização do ClamEB_User

7.5. CLAMEB_GUI

O ClamEB_GUI nada mais é que a interface gráfica. Ele que se responsabiliza por toda a interação com o usuário, recebendo todos os seus comandos e repassando ao mesmo todas as informações que obtém.

Para se realizar o escaneamento de arquivos, seja manualmente ou no momento de acesso, é preciso que o ClamEB_GUI interaja com o ClamEB_User. Isto é realizado de duas maneiras: através das *Window Message* e através do uso de áreas de memória compartilhadas.

Para se fazer uso das *Window Message* foi criado uma mensagem personalizada para o próprio ClamEB, batizada de WM_CLAMEB, de modo a evitar seu uso acidental por outro programa qualquer. A tabela abaixo descreve o significado dessa mensagem de acordo com os parâmetros passados:

wParam	lParam	Significado
0	*Handle* da janela	Início da Comunicação (quando a comunicação for estabelecida será iniciada a *Filter Listening Thread*)
1	*Id* da *Worker Thread*	Requisição de acesso a arquivo
2	Endereço de contexto da *Worker Thread*	Recusa ao acesso do arquivo
3	Endereço de contexto da *Worker Thread*	Aceitação de acesso ao arquivo
6	-	Requisição do escaneamento manual (início da *Manual Scan Thread*)
7	*Handle* da janela	*Refresh* das informações do escaneamento manual
8	*Handle* da janela	Término do escaneamento manual (fim da *Manual Scan Thread*)

TAB. 7-2: Significado das WM_CLAMEB de acordo com seus parâmetros

Como se pode ver, a WM_CLAMEB é excelente para a comunicação de procedimentso simples entre os dois componentes do ClamEB, como o inicio da comunicação entre os dois componentes ou o término do escaneamento manual.

No entanto, vê-se que é necessário passar informações adicionais em alguns casos. Na requisição de acesso a um determinado arquivo pelo usuário, por exemplo, é necessário transmitir do ClamEB_User para a ClamEB_GUI o nome do arquivo infectado , bem como o nome do vírus infectante. Isto é feito utilizando-se áreas de memória compartilhada.

Quando deseja-se transmitir tais informações, a *Worker Thread* cria uma área de memória compartilhada, armazenando na mesma o nome do arquivo infectado, o nome do vírus infectante e seu endereço de contexto, sendo este último necessário para a reativação desta *thread* que dorme em seguida.

De posse dessas informações, o ClamEB_GUI cria uma mensagem para o usuário informando sobre a infecção neste arquivo pelo vírus descrito e perguntando se ele deseja bloquear ou liberar o acesso a este arquivo. Em seguida, o ClamEB_GUI cria envia uma WM_CLAMEB para o ClamEB_User com um dos parâmetros sendo 2 ou 3 e com o outro sendo o endereço de contexto da Worker Thread em questão.

Para se fazer o escaneamento manual, também é necessária a criação de uma área de memória compartilhada na medida em que a cada *refresh* realizado pelo ClamEB_User sobre as informações de escaneaneamento, é preciso trafegar determinadas informações como o número de arquivos infectados, o tamanho do último arquivo escaneado, se esse arquivo estava infectado ou não, etc.

Além das funções de escaneamento, o ClamEB_User possui as funcionalidades de atualização da base de dados e de configuração das opções.

A atualização da base de dados é realizada através da execução direta do *freshclam*, cuja saída é direcionada para a tela gráfica.

A manutenção das informações contidas nas opções é realizada armazenado-as no Registro do Windows. Essas informações são armazenadas na chave `HKEY_LOCAL_MACHINE/SOFTWARE/CLAMEB`, sendo os valores descritos na tabela abaixo:

Nome do valor	Significado
`path`	Diretório de instalação do ClamEB
`version`	Versão do ClamEB
`language`	Idioma atual do ClamEB
`lastDBUpdate`	Última atualização válida da base de dados
`dbpath`	Localização da base de dados
`filterlevel`	Nível do filtro (ainda não implementado)
`onAccessTimeOut`	Tempo de espera da notificação residente

TAB. 7-3: Valores descritos na chave do registro relativa ao ClamEB

8. GUIA DO USUÁRIO

8.1. INSTALAÇÃO

No capítulo anterior, foi descrito todos os detalhes a cerca da arquitetura e do funcionamento do ClamEB. Nesse capítulo, o protótipo será abordado do ponto de vista do usuário, sendo apresentadas todas as suas funcionalidades passo a passo. Ao contrário dos outros capítulos, este não requer nenhum conhecimento especifico de computação sendo totalmente direcionado ao usuário.

Nessa primeira seção, será abordado todos os passos necessário para se instalar o ClamEB.

O primeiro passo para realizar a instalação é dar um duplo clique no instalador:

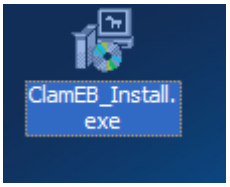

FIG. 8-1: Ícone do Instalador do ClamEB

Em seguida será aberta uma tela, pedindo para informar o local de instalação do ClamEB.

Por padrão, o local de instalação é “C:\Arquivos de Programa\ClamEB”. Pode-se alterar o local de instalação, editando diretamente a linha contendo o caminho completo do diretório de instalação ou clicando no botão “Selecionar” para escolher o local apropriadamente.

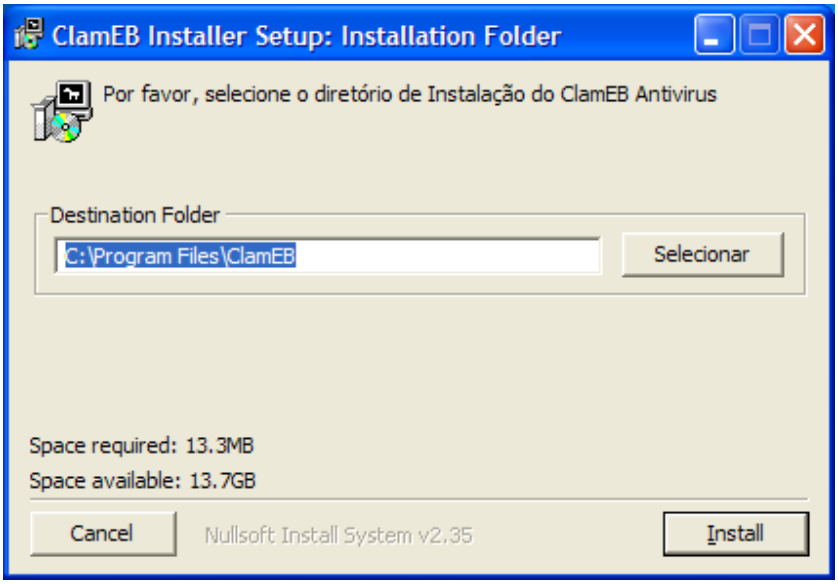

FIG. 8-2: Tela de escolha do diretório de instalação

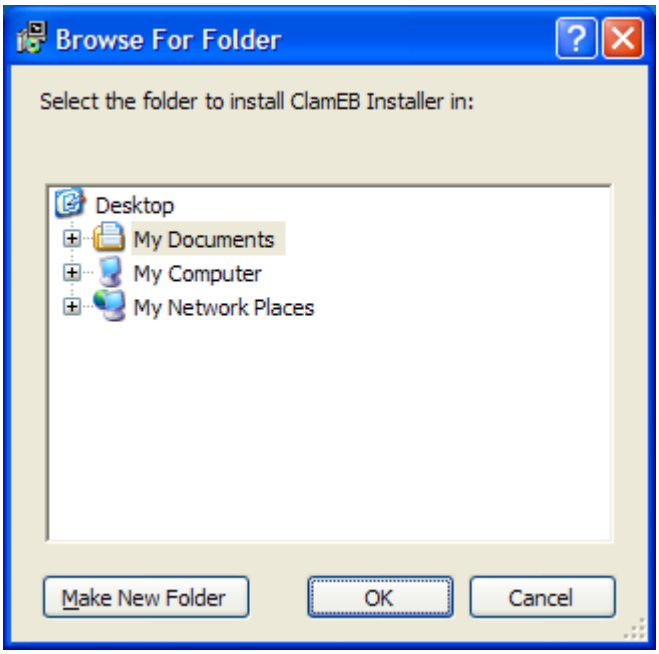

FIG. 8-3: Escolhendo um outro diretório de instalação

Para iniciar a instalação, na tela de escolha do diretório clique no botão “Install”. Será apresentada a seguinte tela:

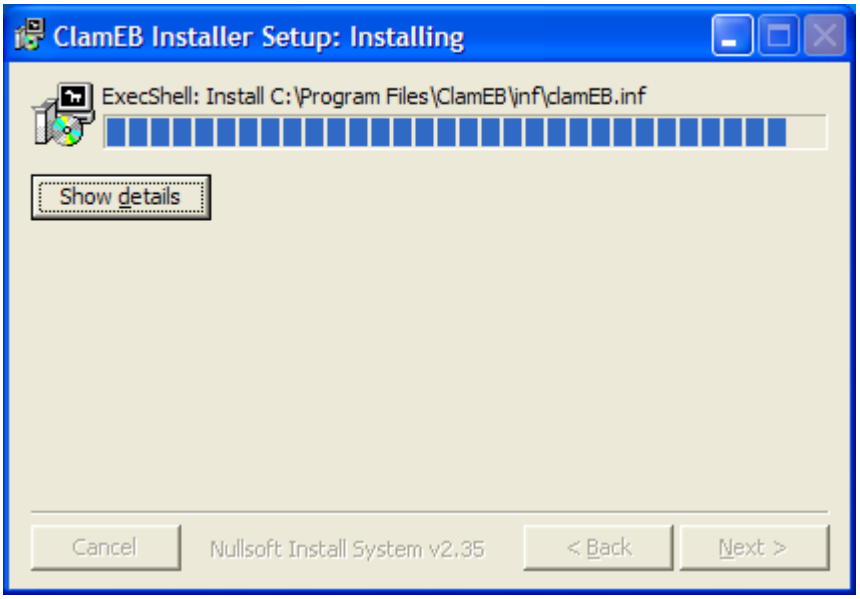

FIG. 8-4: Instalação propriamente dita do ClamEB

Conforme está mostrado na figura acima, o último passo do instalador é a execução de um arquivo do tipo INF que instalará o *driver* no Windows. Pode acontecer de essa execução demorar e, com isso, o botão “Next” do instalador ser habilitado antes. Se isso acontecer, aguarde alguns instantes até clicar nesse botão.

Ao clicá-lo, a seguinte mensagem será exibida ao usuário, perguntado se o usuário deseja reiniciar o computador:

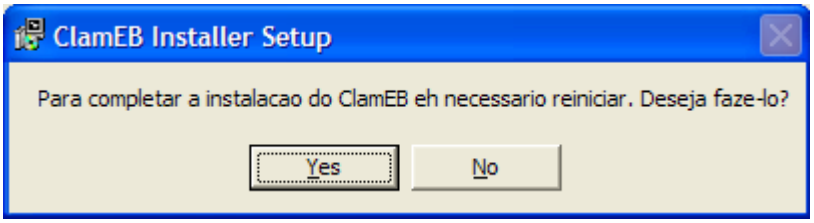

FIG. 8-5: Mensagem da instalação pedindo para se reiniciar o computador

É importante ressaltar que, embora o instalador permita que o usuário não reinicie o computador, o anti-vírus não funcionará corretamente até que isto seja feito.

Agora, quando o computador for reiniciado, o anti-vírus estará pronto para ser executado com todas as suas funcionalidades presentes.

8.2. FUNCIONALIDADES PRINCIPAIS

Decorrida a instalação, o ClamEB será configurado para ser carregado durante a inicialização do Windows. Quando o ClamEB está carregado um ícone na forma de escudo fica localizado ao lado do relógio conforme mostrado na figura abaixo:

FIG. 8-6: Ícone do ClamEB ao lado do relógio

Na figura abaixo, temos a tela principal do ClamEB,. O *design* do ClamEb foi projetado de modo que as funcionalidades pudessem ser acessadas rapidamente e a interface fosse o mais simples possível. Assim, ao decorrer desse capitulo, pode-se notar que não existem muitas telas desse anti-vírus, exatamente pelo fato dessa simplicidade ser escolhida como atributo da GUI.

FIG. 8-7: Tela inicial do ClamEB

Na tela principal, podemos distinguir quatro das cinco funcionalidades do anti-vírus. São elas:

- **Escaneamento completo de todos os dispositivos locais**, i.e., o escaneamento dos HDs locais, dos CDs ou DVDs que estiverem nos *drives* e dos *pendrives* que estiverem conectado no computador.

- **Escaneamento parcial de diretórios ou arquivos específicos**;

- **Atualização da base de dados**;

- **Alteração dos parâmetros de opções;**

Além dessas quatro funcionalidades, existe uma quinta que é a funcionalidade de **residência do anti-vírus**. A medida que os arquivos são executados pelos programas no Windows, o ClamEB verifica se eles estão infectados e, em caso positivo, antes de se completar o acesso ao arquivo, o anti-vírus pergunta ao usuário se ele deve permitir que o programa acesso o arquivo ou não.

8.3. ESCANEAMENTO COMPLETO

Conforme dito antes, o escaneamento completo é uma funcionalidade que permite todos os arquivos de todos os dispositivos locais disponíveis.

Para fazê-lo, clique na guia “Escaneamento Completo” e nela clique no botão “Escanear agora”. Para abrir a guia “Escaneamento completo”, pode-se também clicar com o botão direito no escudo ao lado do relógio e clicar em “Abrir o ClamEB Antivirus”, ou então dar um duplo clique no ícone ao lado do relógio em forma de escudo.

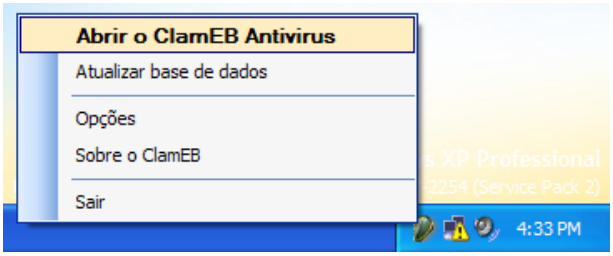

FIG. 8-8: Opção “Abrir o ClamEB Antivirus” no menu de contexto do ícone ao lado do relógio

Será apresentada a seguinte tela:

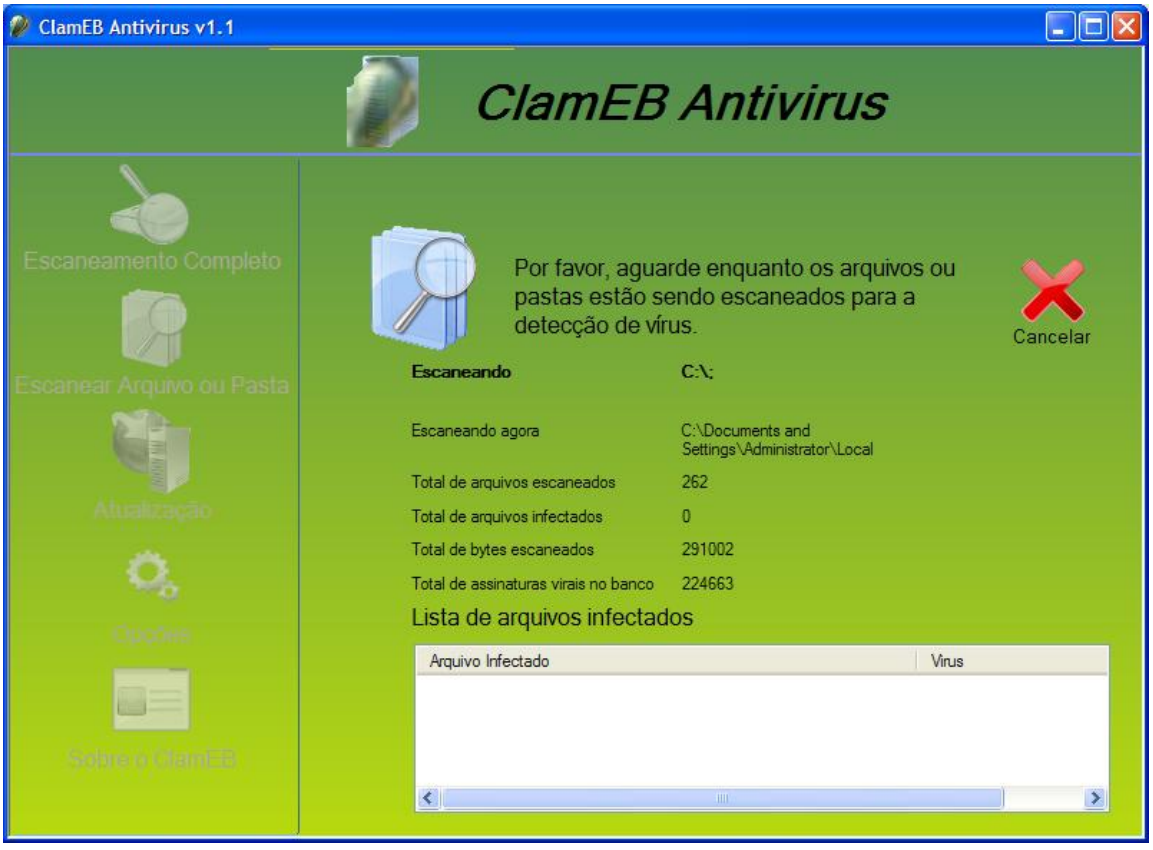

FIG. 8-9: Escaneando os dispositivos locais

Durante o escaneamento todos os arquivos dos dispositivos locais serão verificados se estão ou não contaminados. Em caso positivo, o arquivo será listado na “Lista de arquivos infectados” com o nome do vírus infectante ao seu lado.

Enquanto os arquivos estiverem sendo escaneados, não será possível acessar nenhuma outra funcionalidade do sistema.

Durante todo o processo de escaneamento, pode-se clicar no botão “Cancelar” e interromper a operação.

Quando o processo de escaneamento finalizar, todos os botões relativos a todas as funcionalidades serão habilitados novamente, conforme mostrado na figura abaixo:

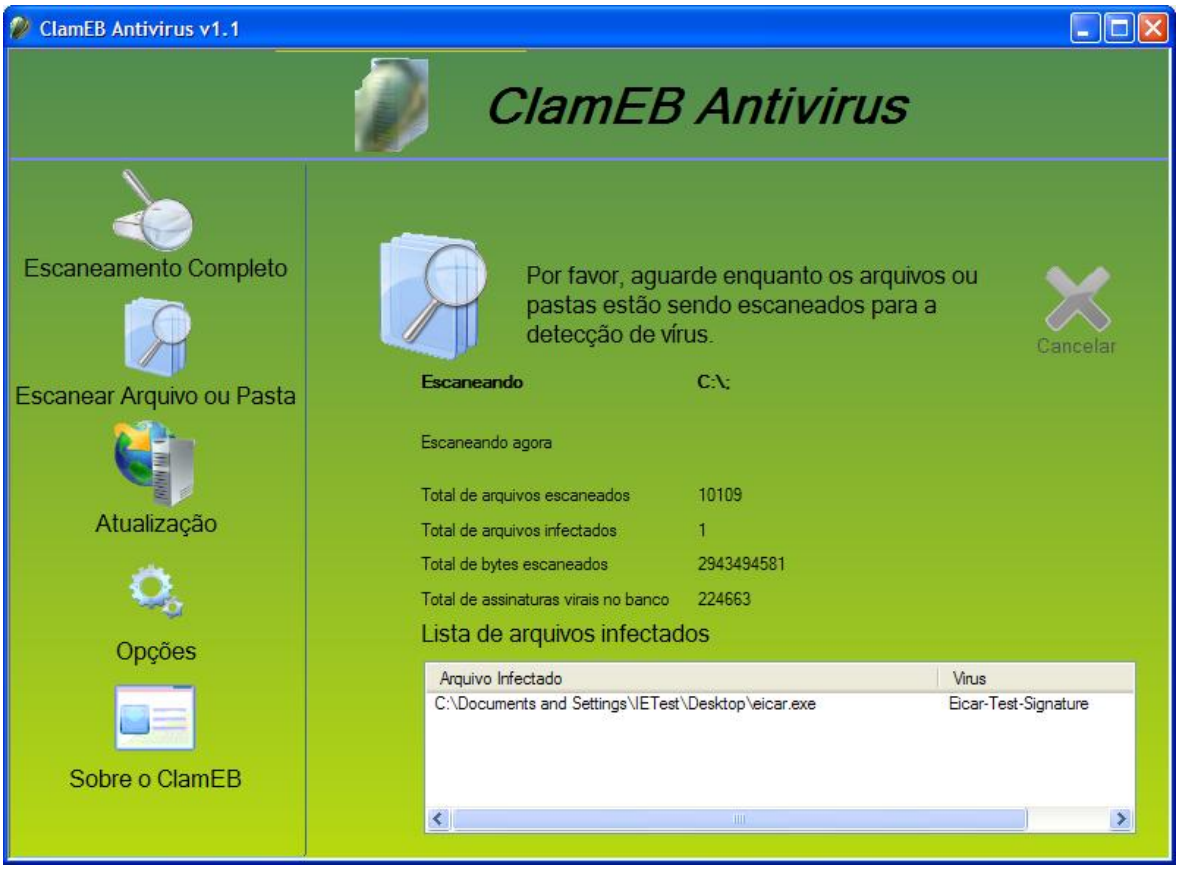

FIG. 8-10: Fim do processo de escaneamento completo

8.4. ESCANEAMENTO PARCIAL

Para realização de um escaneamento parcial de um diretório ou de arquivos específicos, utiliza-se a funcionalidade de escaneamento parcial.

Clique na guia “Escanear Arquivo ou Pasta”. Será apresentada a seguinte tela:

FIG. 8-11: Tela de escaneamento parcial

Nesta tela, pode-se selecionar para escanear ou arquivos ou uma pasta de arquivos. Para informar a localização dos arquivos ou da pasta, pode-se editar manualmente no campo disponível ou então clicar no botão ao lado para procurar o arquivo ou a pasta que se deseja escanear.

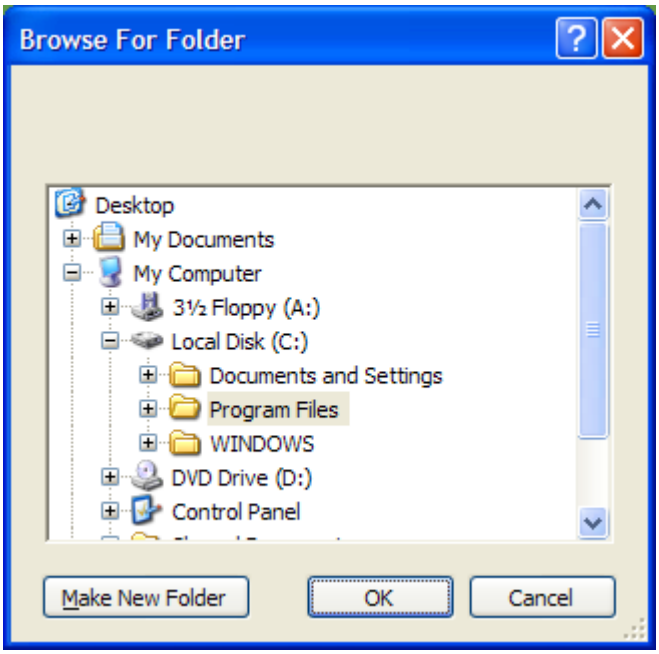

FIG. 8-12: Procurando uma pasta para se escanear

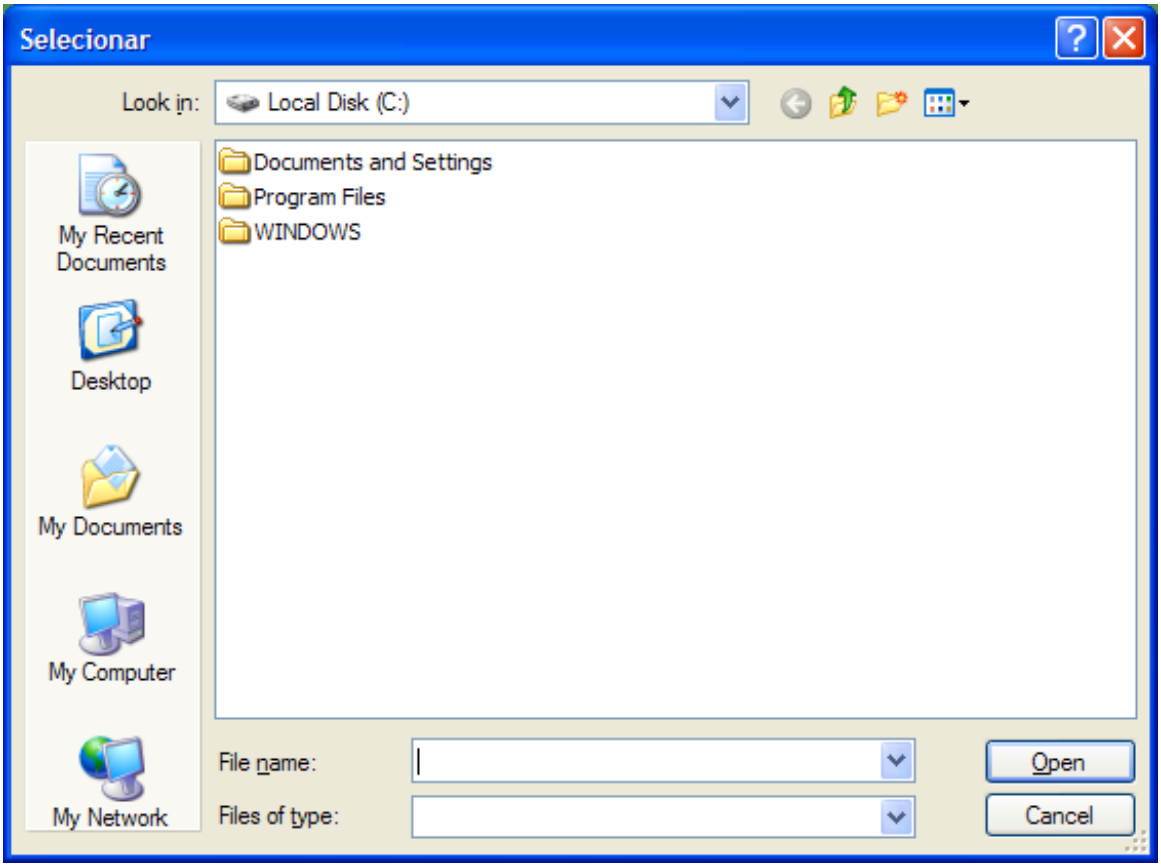

FIG. 8-13: Procurando arquivos para serem escaneados

Selecionados os arquivos ou pasta para serem escaneados, clique no botão “Escanear agora” para iniciar o escaneamento parcial. A tela de escaneamento parcial é a mesma que a de escaneamento completo, sendo todo o processo exatamente igual.

FIG. 8-14: Escaneando arquivos ou pasta

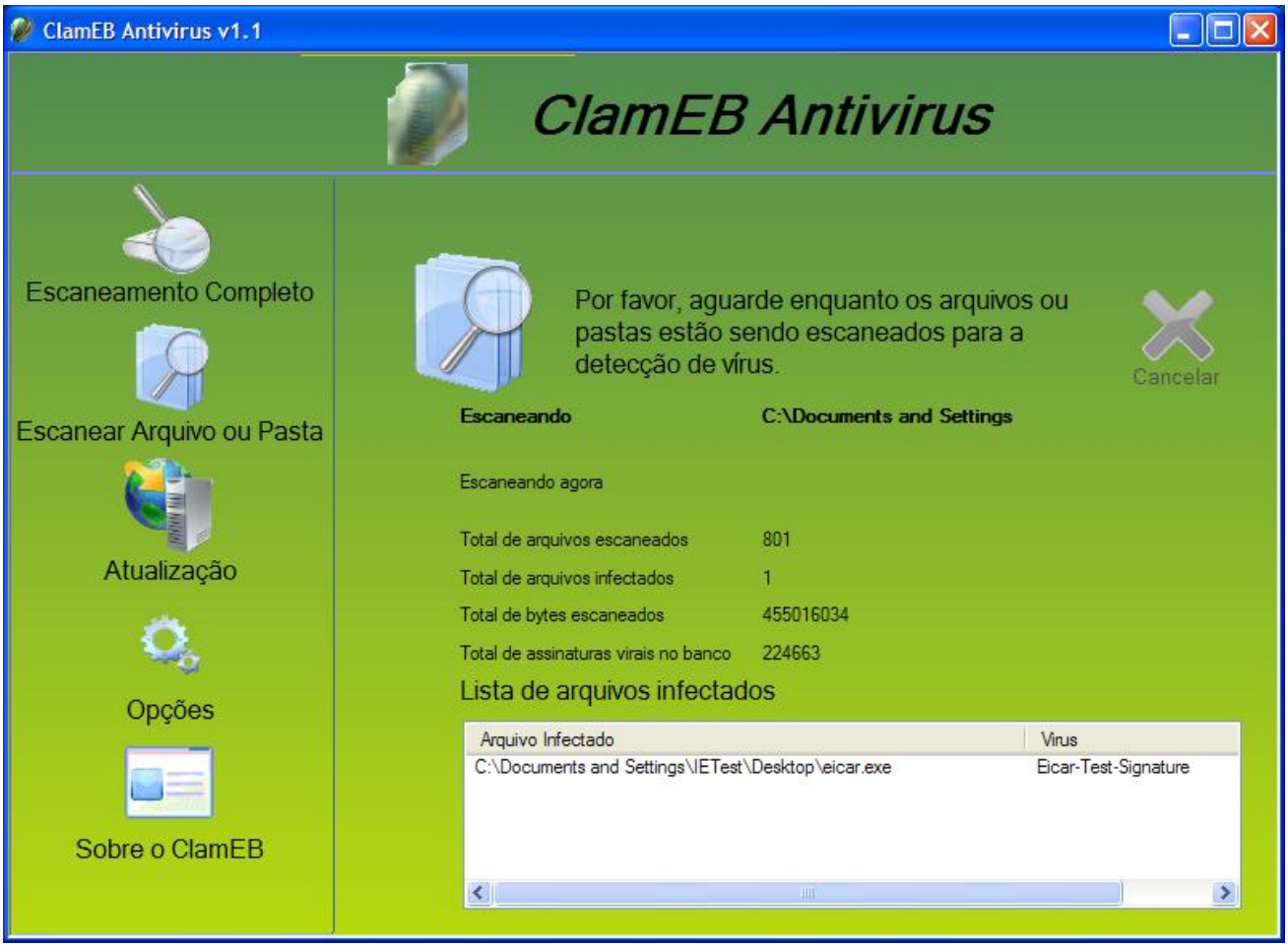

FIG. 8-15: Fim do escaneamento parcial

8.5. ATUALIZAÇÃO DA BASE DE DADOS

Para atualizar a base de dados, clique na guia “Atualização”. Pode-se alternativamente clicar com o botão direito no ícone em forma de escudo ao lado do relógio e, no menu de contexto subseqüente, clicar em “Atualizar base de dados”.

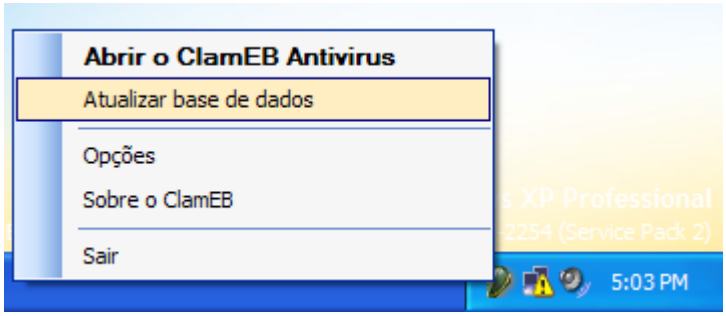

FIG. 8-16: Opção “Atualizar base de dados” do menu de contexto do ícone ao lado do relógio

Em seguida, será apresentada a seguinte tela:

FIG. 8-17: Tela de atualização da base de dados

Para começar a atualização da base de dados, clique no botão “Atualizar base de dados”.

FIG. 8-18: Realizando o *downloading* da base de dados para atualizar a mesma

Durante esse processo, assim como no escaneamento, não se pode acessar nenhuma outra funcionalidade do sistema, devendo-se esperar o processo de atualização terminar para que as guias das funcionalidades sejam novamente habilitadas.

8.6. OPÇÕES

Esta funcionalidade permite editar determinadas configurações de funcionamento do ClamEB. Na versão corrente em que se está escrevendo esta monografia, há três parâmetros que podem ser editados:

- **Localização da base de dados**, i.e., o caminho completo do diretório onde se encontra a base de dados do ClamEB;

- **Idioma em que se encontra o anti-vírus**;

- ***Timeout* da notificação residente**, i.e., o tempo de espera pela resposta do usuário para a requisição de acesso aos arquivos infectados.

Para entrar na tela de opções, selecione a guia “Opções” na tela principal, ou então clique com o botão direito no ícone do escudo ao lado do relógio e selecione o item “Opções” no menu de contexto.

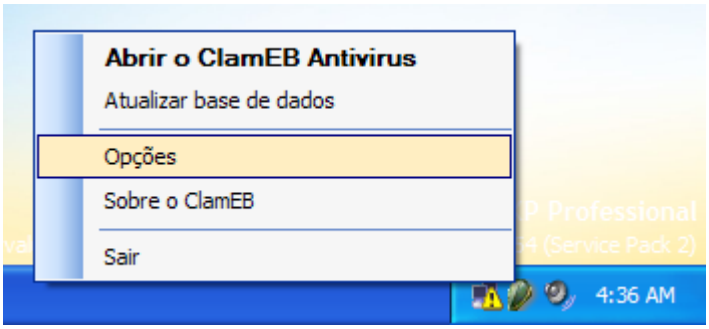

FIG. 8-19: Opção “Opções” no menu de contexto do ícone ao lado do relógio

Será aberta a seguinte tela:

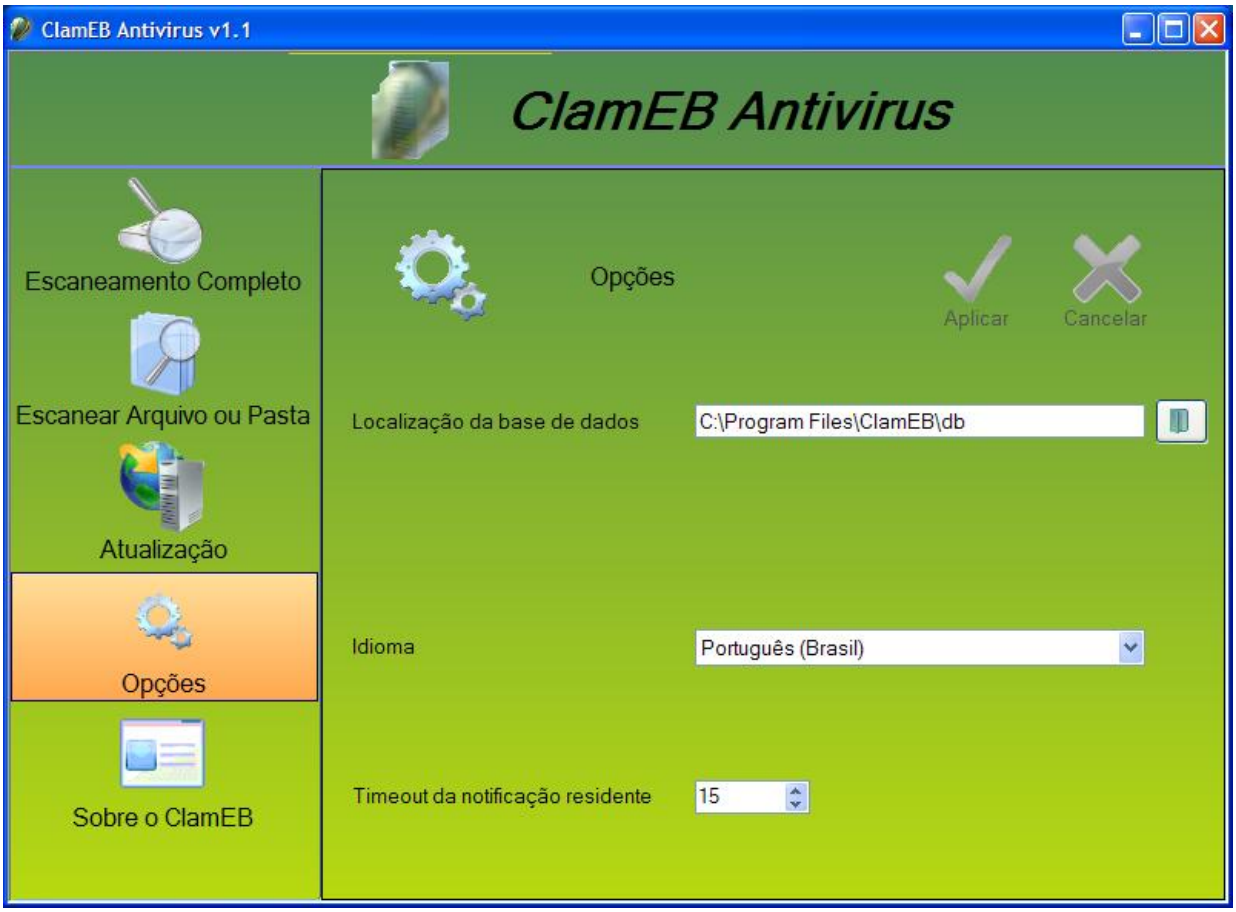

FIG. 8-20: Tela de opções

Nesta tela, pode-se alterar qualquer um dos parâmetros citados anteriormente. Toda vez que um parâmetro for alterado, nesta tela os botões “Aplicar” e “Cancelar” serão habilitados,

permitindo validar ou cancelar as alterações feitas nesses parâmetros. As alterações dos parâmetros das opções quando validadas são persistidas no registro do Windows. Para mais detalhes, consulte o capítulo sobre a arquitetura do sistema.

É válido lembrar que todas as alterações feitas nesses parâmetros só farão efeito da próxima vez que o programa for iniciado. Assim, se, por exemplo, o idioma do anti-vírus for trocado de português para inglês, o programa só será traduzido para inglês na próxima vez que for iniciado, permanecendo em português até o mesmo ser finalizado.

8.7. NOTIFICAÇÕES RESIDENTES

Conforme dito anteriormente, há uma quinta funcionalidade do ClamEB que não aparece nas guias da tela principal. Trata-se da funcionalidade de residência do anti-vírus.

Além de o usuário poder escanear os arquivos do computador automaticamente, o ClamEB Antivirus possui um *resident shield* que permite o escaneamento automático de arquivos do sistema operacional a medida que os mesmos são acessados, sem nenhuma necessidade de intervenção do usuário.

Esse mecanismo é chamado de escaneamento *on-access* e é mais um mecanismo de proteção que os anti-vírus oferecem ao computador.

Toda vez que um arquivo infectado for usado pelo usuário ou por algum programa, o ClamEB enviará uma mensagem ao usuário dizendo que esse determinado arquivo está infectado por um determinado vírus e perguntando se o usuário deseja bloquear seu acesso ou permití-lo.

A figura abaixo exemplifica essa mensagem:

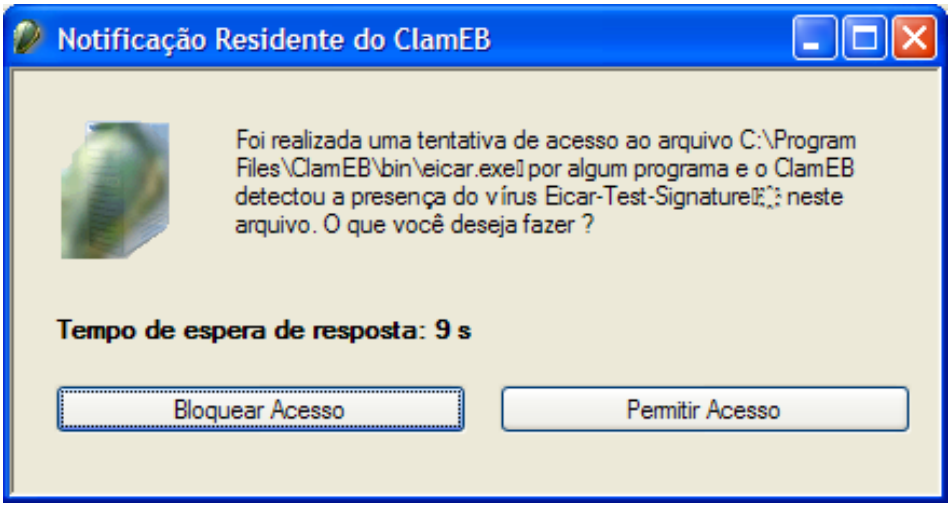

FIG. 8-21: Exemplo de notificação residente

Essa mensagem tem um tempo de espera de resposta para se encerrar. No final desse tempo, se o usuário não escolher entre permitir o acesso ao arquivo ou bloqueá-lo, o anti-vírus por padrão decidirá bloquear o acesso ao arquivo.

A figura abaixo mostra um exemplo de arquivo executável, cuja execução fora bloqueada pelo anti-vírus:

FIG. 8-22: Exemplo de bloqueio de acesso ao arquivo

9. CONCLUSÃO

9.1. RESULTADOS ALCANÇADOS

Com este trabalho, conseguimos construir com sucesso um protótipo de anti-vírus estático livre e residente para ambiente Windows, que cumpre perfeitamente os principais requisitos funcionais de um anti-vírus de mercado.

As seguintes funcionalidades foram implementadas no protótipo construído:

- Instalador gráfico, com possibilidade de escolha do local onde será instalado o anti-vírus;
- Interface gráfica com o usuário prática e fácil de usar;
- Escaneamento completo do sistema;
- Escaneamento parcial do sistema;
- Escaneamento *on-access*, com notificações residentes em caso de infecção viral, que permitem ao usuário escolher se ele quer realmente acessar o arquivo ou não;
- Possibilidade de atualização da base de dados com as definições dos vírus, utilizando a base de dados do ClamAv, que é atualizada frequentemente pela comunidade associada;
- Suporte multi-línguas, permitindo a escolha entre 4 línguas (português, inglês, espanhol e italiano);

O anti-vírus desenvolvido é tão confiável quanto é o próprio ClamAv. Conforme visto anteriormente, a comunidade tem comprovado que ele é tão ou mais confiável que os demais anti-vírus disponíveis no mercado. Outra vantagem do anti-vírus construído nesse trabalho é que ele é totalmente livre, podendo, portanto, ser implantado nas redes corporativas do Exército Brasileiro e outras instituições, sem resultar em nenhum ônus financeiro para isso.

Além disso, com este trabalho foi possível agregar um conhecimento considerável sobre assuntos pouco conhecidos e importantes no contexto da Segurança de Informação, que, conforme vimos, é uma área de interesse estratégico para o Exército Brasileiro. Esse conhecimento foi devidamente documentado e poderá servir de suporte para atividades de desenvolvimento futuro nessa área.

9.2. POSSÍVEIS MELHORIAS

Na forma atual em que se encontra o nosso protótipo de anti-vírus, feitas as devidas ressalvas de desempenho, por se tratar justamente de um protótipo, ele já está plenamente

funcional e pode ser utilizado como anti-vírus em estações de trabalho rodando sistemas operacionais Windows.

Ainda existem, no entanto, funcionalidades que podem vir a ser implementadas no futuro, agregando mais valor a este produto.

Dentre as possíveis melhorias, podemos citar:

- Atualização automática – essa melhoria permitirá ao anti-vírus se atualizar de forma automática, sem necessitar da intervenção do usuário. Isso tornará o sistema mais confiável, na medida em que sua base de dados dos vírus conhecidos estará sempre atualizada, permitindo uma melhor proteção à máquina do usuário;
- Reestruturação dos arquivos CVD – essa melhoria trata da reorganização dos dados contidos nos arquivos CVD usando outras estruturas de dados, de forma a facilitar e melhorar o desempenho do seu acesso;
- Técnicas de análise dinâmica de *malware* – essa melhoria permitirá ao anti-vírus detectar vírus por meio da análise do comportamento do programa no ambiente do sistema operacional, tornando este um anti-vírus dinâmico e lidando melhor com os vírus polimórficos;
- *Cache* dos arquivos acessados pelo sistema operacional – essa melhoria otimizará o desempenho na medida em que, mantendo-se um *cache* do status dos arquivos recentemente verificados, não será necessário inspecioná-los de novo, em caso de uma nova solicitação de acesso, utilizando-se ao invés disso o resultado gravado no *cache*. Claro que isso acarretaria também uma preocupação em como manter tal *cache* válido, de acordo com as alterações feitas nos arquivos;
- Integração do ClamEB com o menu de contexto do *Windows Explorer* – essa melhoria permitirá ao usuário escanear mais facilmente qualquer arquivo do sistema, bastando para isso utilizar uma opção no menu de contexto obtido clicando-se com o botão direito no arquivo alvo;
- Integração do ClamEB com os clientes de *email*, como o Gmail – essa melhoria permitirá ao usuário ter os arquivos obtidos via clientes de *emails* automaticamente escaneados, garantindo uma proteção constante contra vírus transmitidos por esse meio;

- Teste e melhoria do desempenho – de maneira geral, o anti-vírus ainda precisa ter seu desempenho geral avaliado e melhorado antes de ser utilizado em larga escala nas estações de trabalho do Exército Brasileiro.

As funcionalidades acima descritas podem inclusive vir a gerar novos temas de Projeto de Fim de Curso ligados ao Grupo de Segurança da Informação.

9.3. CONSIDERAÇÕES FINAIS

O anti-vírus é hoje um *software* indispensável em qualquer ambiente computacional, principalmente nos baseados em plataforma Windows. Nesse sentido, o presente trabalho logrou êxito em criar um produto extremamente útil e necessário, tanto no meio militar como no meio civil.

Além disso, este trabalho demonstra claramente que, com o material humano bem qualificado de que dispomos, com esforço e dedicação, é possível adquirirmos *expertise* em áreas de conhecimento que o senso comum diria muito complexas, restritas ou de difícil acesso, não deixando em nada a desejar com relação aos profissionais de computação de outros países.

10. REFERÊNCIAS BIBLIOGRÁFICAS

ANDERSSON, K. “Turing Machines and Undecidability with Special Focus on Computer Viruses.” 2003.
BIDGOLI, Hossein. *Handbook of Information Security: Threats, Vulnerabilities, Prevention, Detection, and Management.* John Wiley & Sons, 2006.
ClamWin. *ClamWin.* www.clamwin.com (acesso em 2007/8).
COHEN, Fred. “Computer viruses: Theory and Experiments.” *ACM Computers and Security,* 1987: 22-35.
European Institute for Computer Antivirus Research. *EICAR.* http://www.eicar.org/ (acesso em 2007/8).
HONEYCUTT, Jerry. *Microsoft Windows Registry Guide.* 2 Ed. Redmond: Microsoft Press, 2005.
Microsoft Corporation. *MSDN.* msdn2.microsoft.com/en-us/default.aspx (acesso em 2007/8).
OGNESS, John. “Dazuko: An Open Solution to Facilitate 'On-Access' Scanning.” *Virus Bulletin Conference.* 2003.
ORWICK, Penny, e Guy SMITH. *Developing Drivers with the Windows Driver Foundation.* Redmond: Microsoft Press, 2007.
SOLOMON, David, e Mark RUSSINOVICH. *Microsoft Windows Internals.* 4 Ed. Redmond: Microsoft Press, 2005.
Sourcefire. *ClamAv.* www.clamav.net (acesso em 2007/8).

11. APÊNDICES

11.1. INSTALAÇÃO DO WINDOWS DRIVER KIT

O site http://www.microsoft.com/whdc/DevTools/WDK/WDKpkg.mspx contém todas as instruções necessárias para baixar o *Windows Driver Kit* (WDK). O WDK é disponibilizado sob a forma de um arquivo ISO, devendo ser gravado para um DVD, que será então utilizado na instalação propriamente dita.

Após a confecção do DVD contendo o WDK, coloque-o no *drive* de DVD e execute o arquivo Installer.exe. A seguinte tela será apresentada:

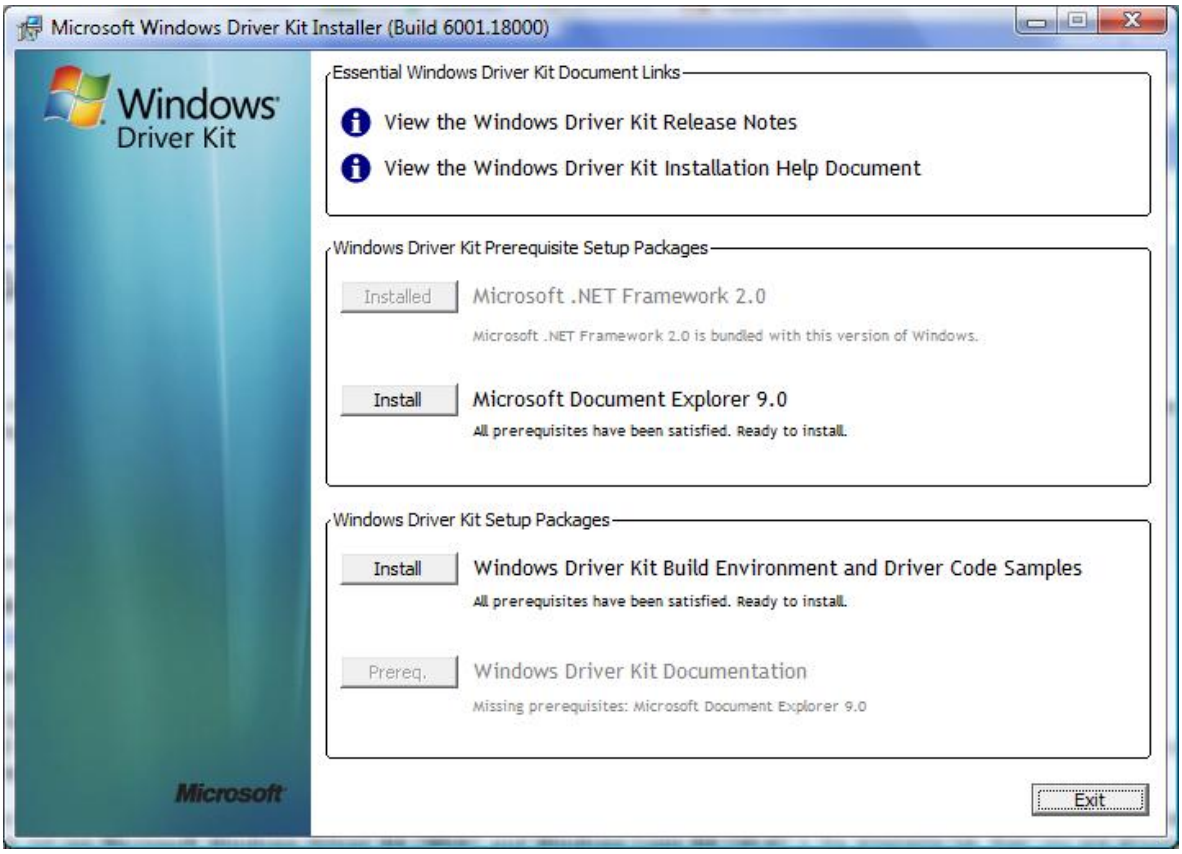

FIG. 11-1: Tela inicial do instalador da WDK

Primeiramente, se a *.NET Framework 2.0* não estiver instalada no Windows, será necessário instalá-la. Para isso, clique instale a opção *Microsoft .NET Framework 2.0* .

Agora, clique na opção de instalação do *Windows Driver Kit Build Environment and Driver Code Samples*. Será apresentada a seguinte tela:

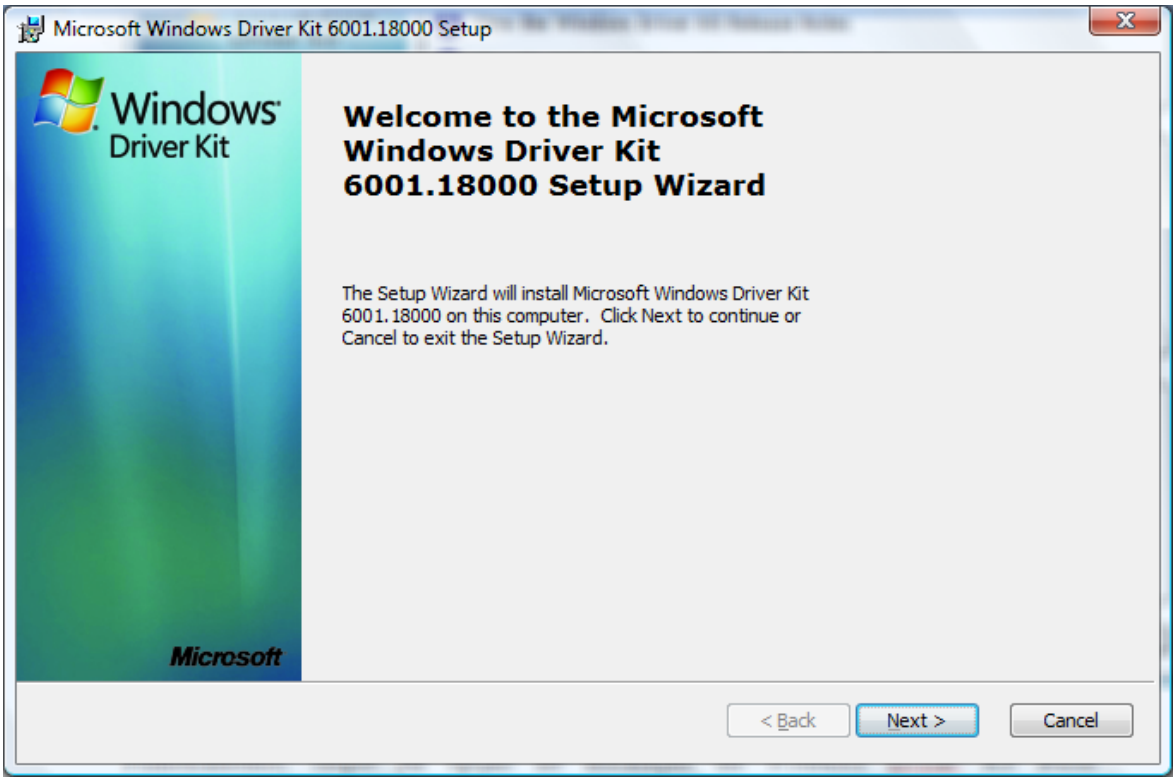

FIG. 11-2: Tela inicial do processo de instalação do *Windows Driver Kit Build Environment and Driver Code Samples*

Clique em *Next* para ir para a seguinte tela:

FIG. 11-3: Tela de termos da licença de uso do *Windows Driver Kit Build Environment and Driver Code Samples*

Leia os termos de licença, escolha a opção de aceitar os termos da licença e clique em *Next*:

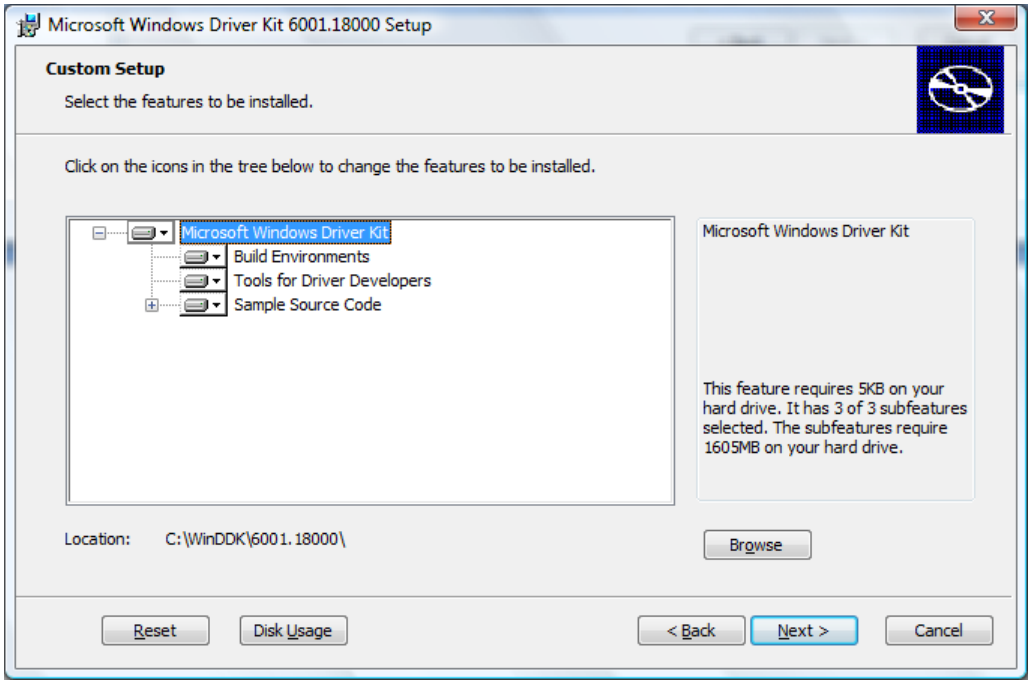

FIG. 11-4: Tela de seleção de quais pacotes do *Windows Driver Kit Build Environment and Driver Code Samples* serão efetivamente instalados

Clique em *Next*, para ser apresentada a seguinte tela:

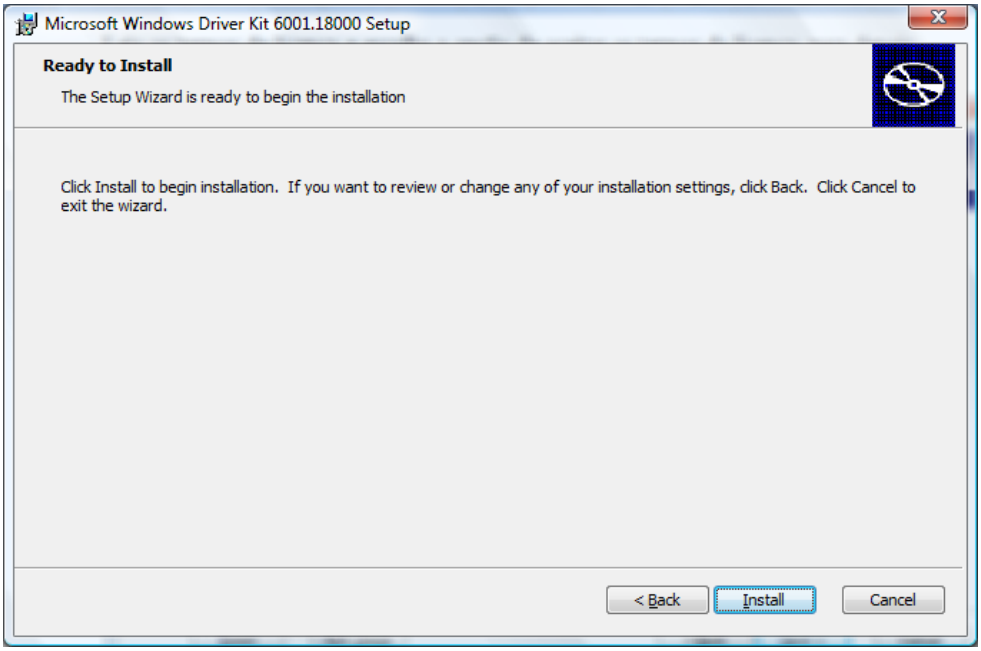

FIG. 11-5: Instalador pronto para iniciar a instalação do *Windows Driver Kit Build Environment and Driver Code Samples*

Clique em *Install* para começar a instalação.

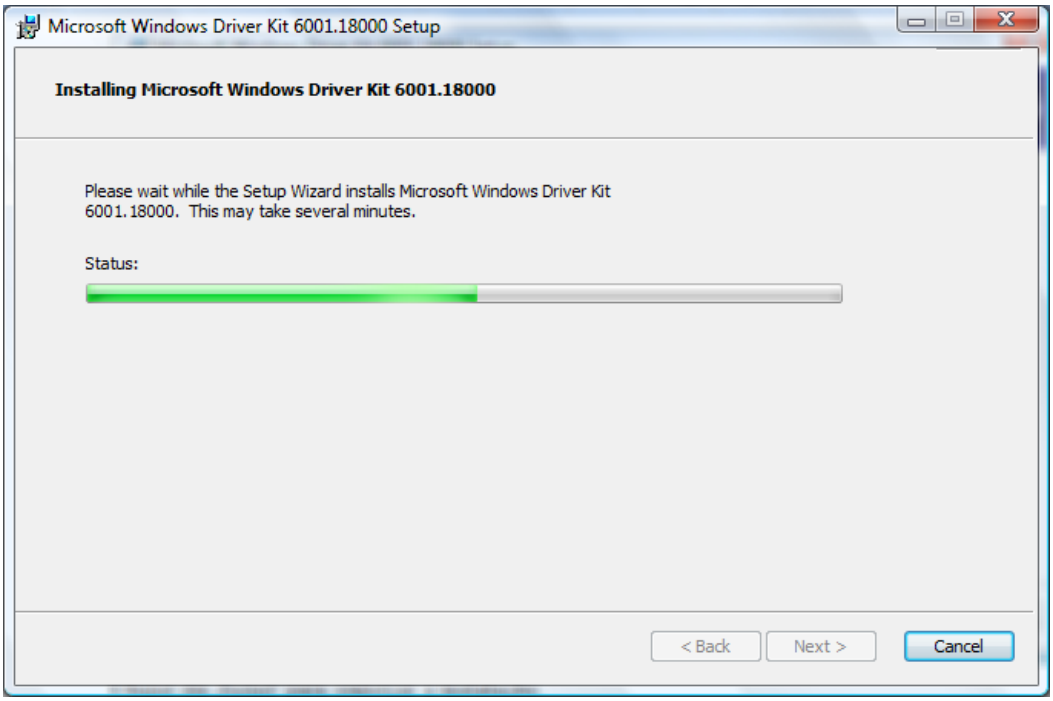

FIG. 11-6: Instalador realizando a instalação do *Windows Driver Kit Build Environment and Driver Code Samples*

Ao completar a instalação será apresentado a seguinte tela:

FIG. 11-7: Instalação do *Windows Driver Kit Build Environment and Driver Code Samples* completa

Agora selecione a segunda opção *Microsoft Document Explorer 9.0* .

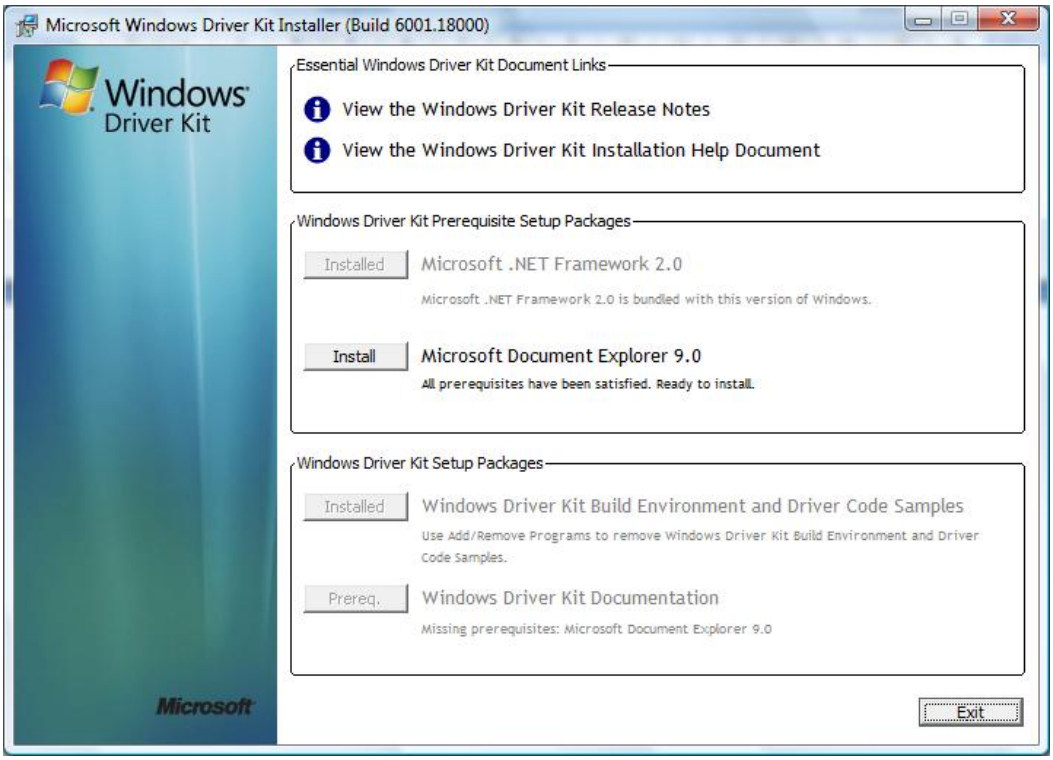

FIG. 11-8: Tela do instalador após a instalação do *Windows Driver Kit Build Environment and Driver Code Samples*

Será apresentada a seguinte tela:

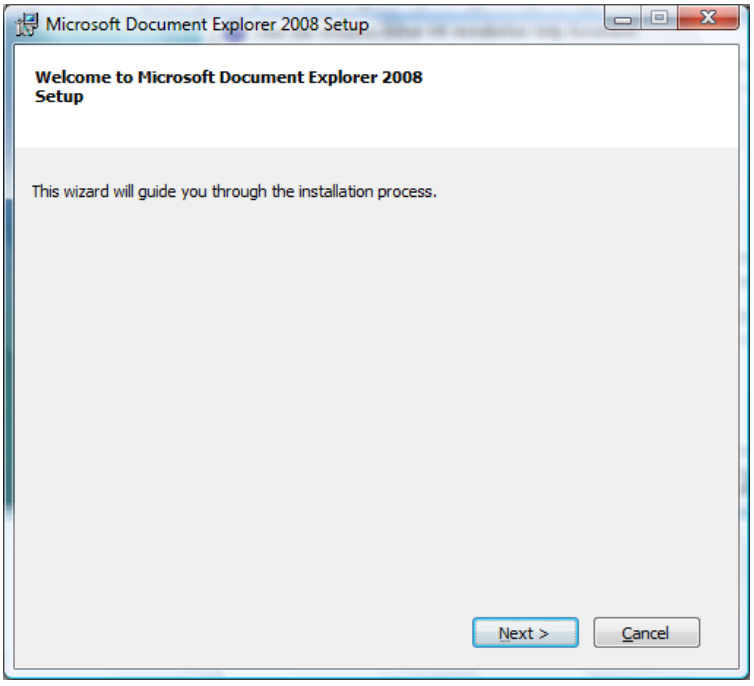

FIG. 11-9: Tela inicial do processo de instalação do *Document Explorer 9.0*

Clique em *Next*, para ser apresentado a seguinte tela:

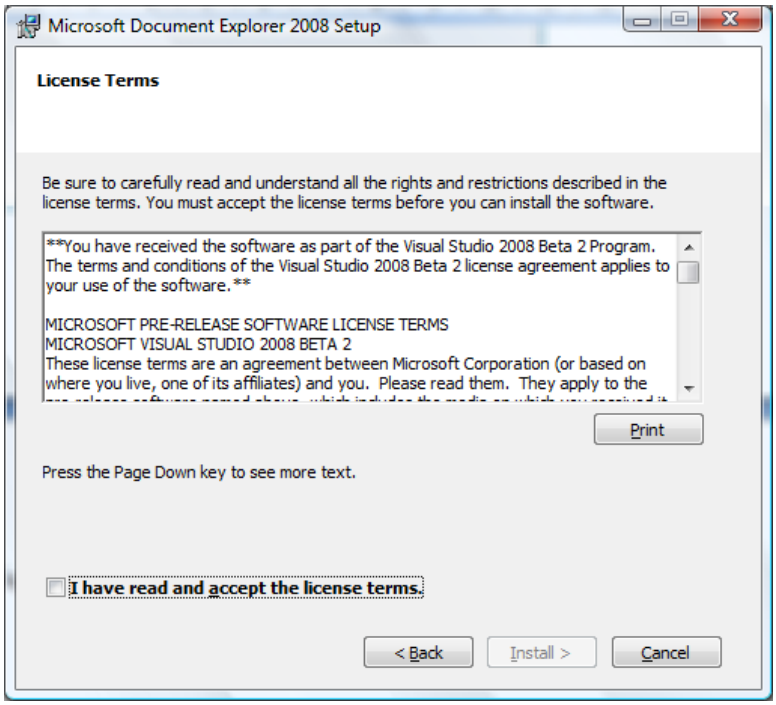

FIG. 11-10: Tela de termos da licença de uso do *Document Explorer 9.0*

Leia os termos de licença, dê um *check* em *I have read and accept the license terms* e clique em *Install* para começar a instalar a documentação:

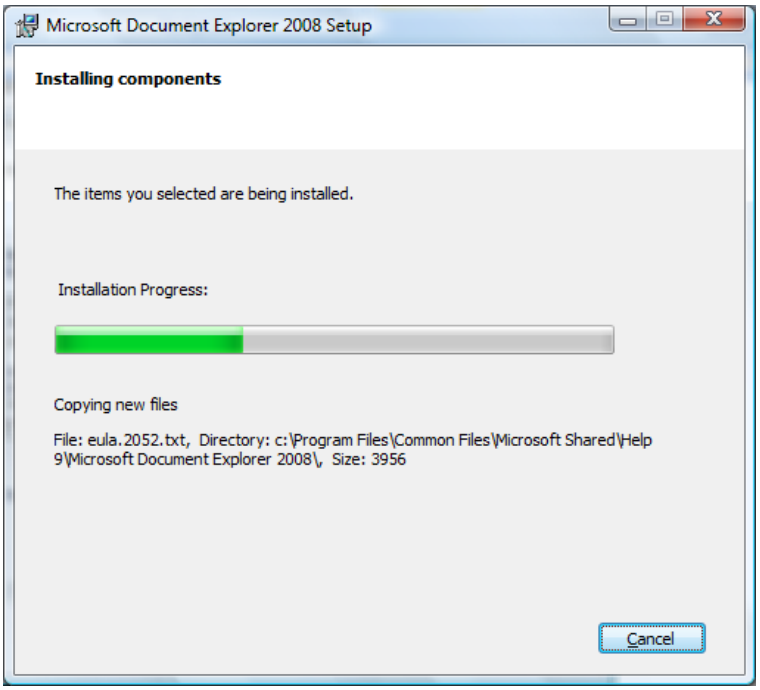

FIG. 11-11: Instalador realizando a instalação do *Document Explorer 9.0*

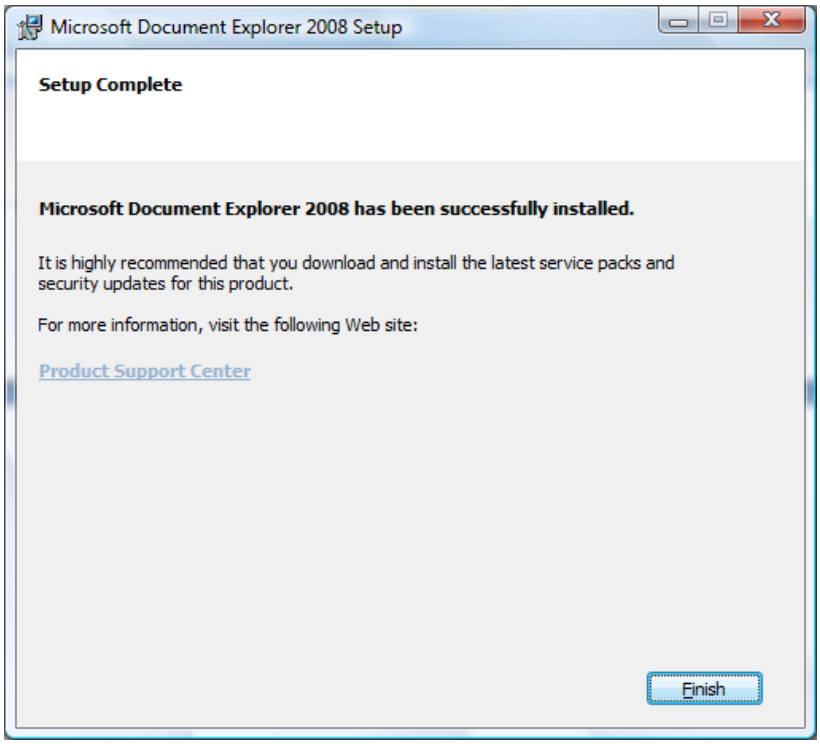

FIG. 11-12: Instalação do *Document Explorer 9.0* completa

Clique em *Finish* para completar a instalação da documentação.

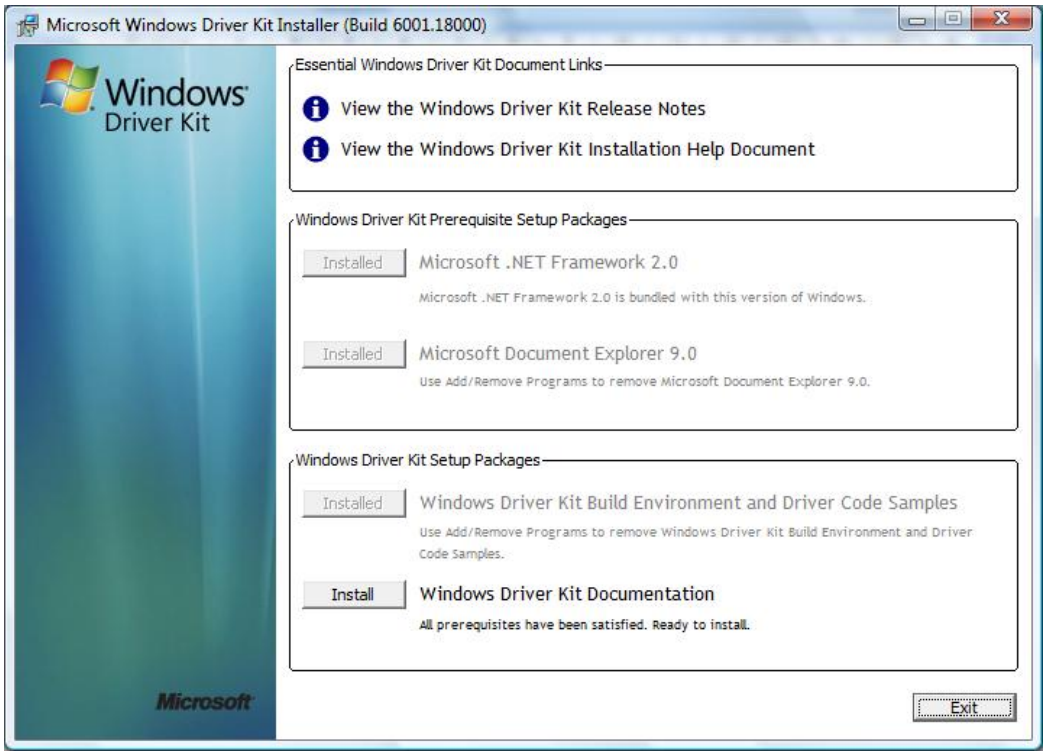

FIG. 11-13: Tela do instalador após a instalação do *Document Explorer 9.0*

Agora selecione a opção *Windows Driver Kit Documentation* e instale.

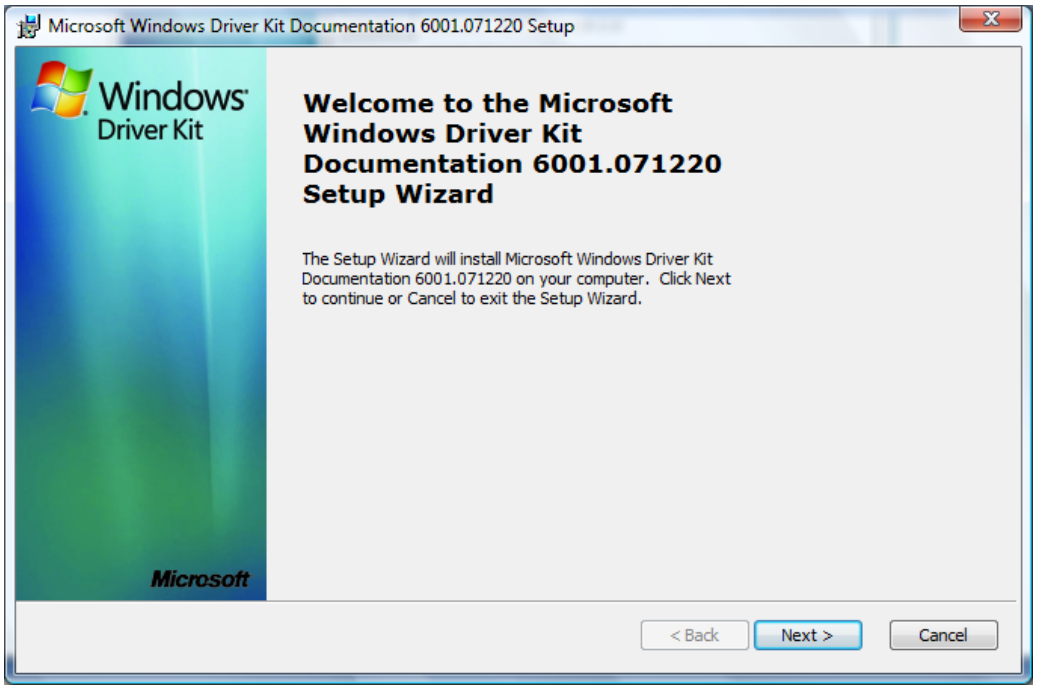

FIG. 11-14: Tela inicial do processo de instalação da *Windows Driver Kit Documentation*

FIG. 11-15: Tela de termos da licença de uso da *Windows Driver Kit Documentation*

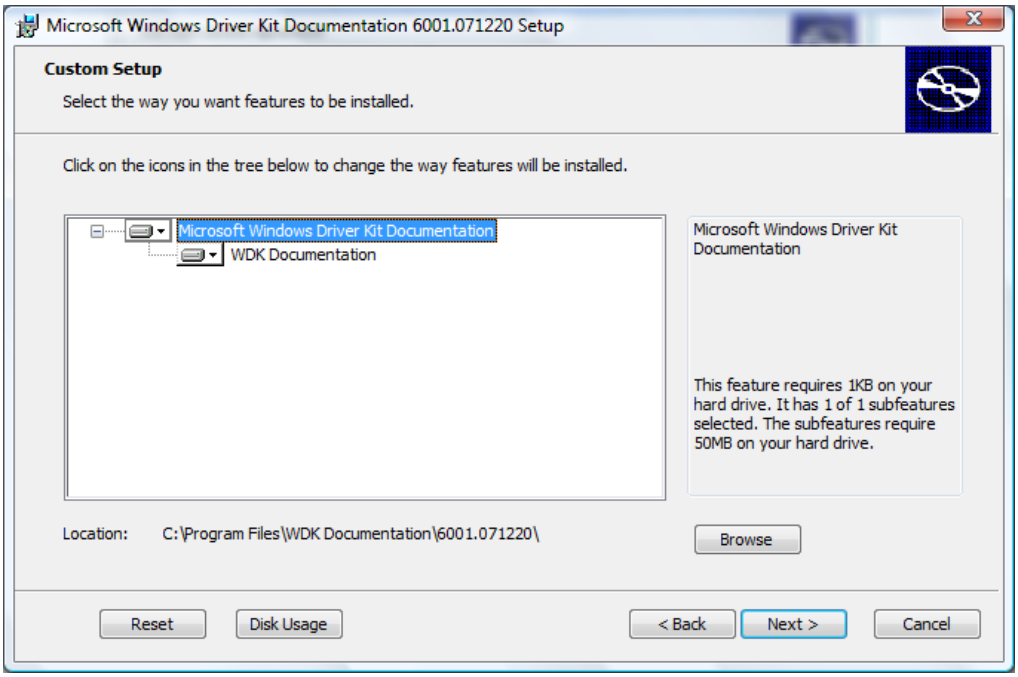

FIG. 11-16: Tela de seleção de quais pacotes da *Windows Driver Kit Documentation* serão efetivamente instalados

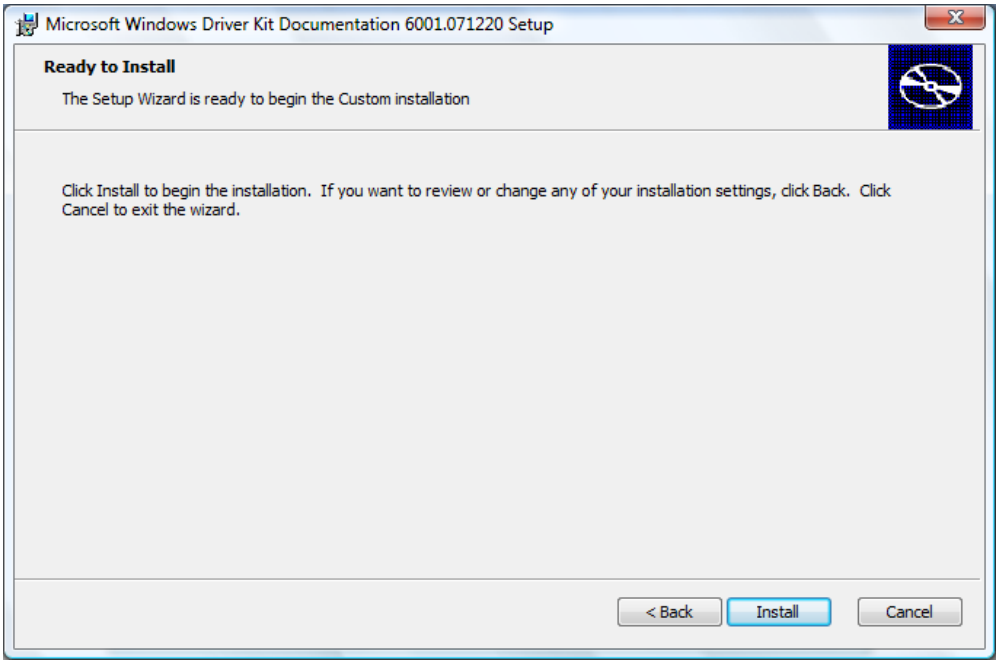

FIG. 11-17: Instalador pronto para iniciar a instalação da *Windows Driver Kit Documentation*

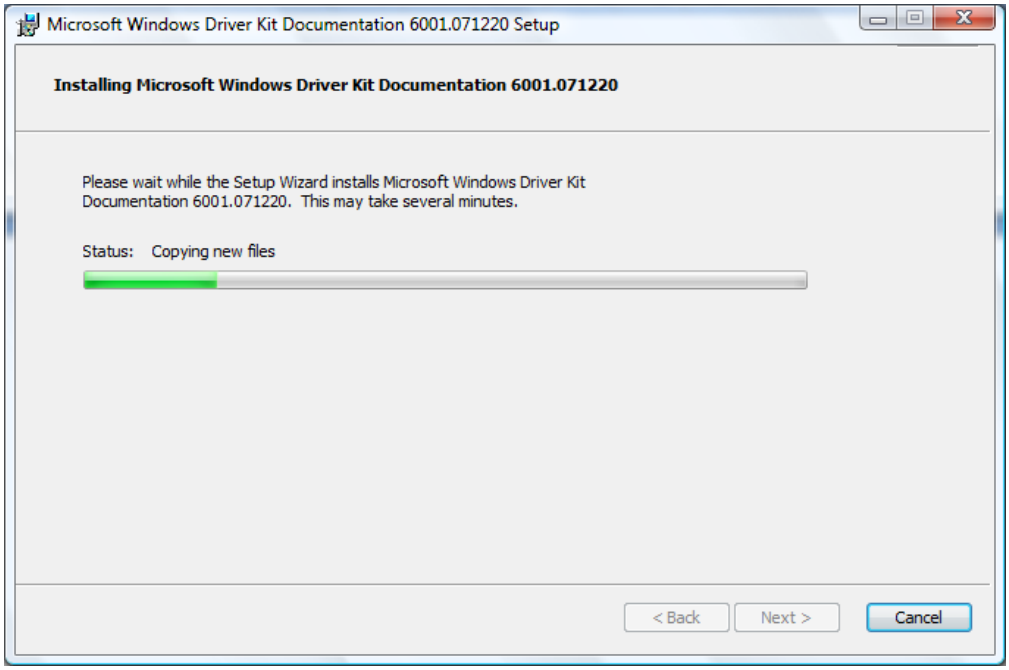

FIG. 11-18: Instalador realizando a instalação da *Windows Driver Kit Documentation*

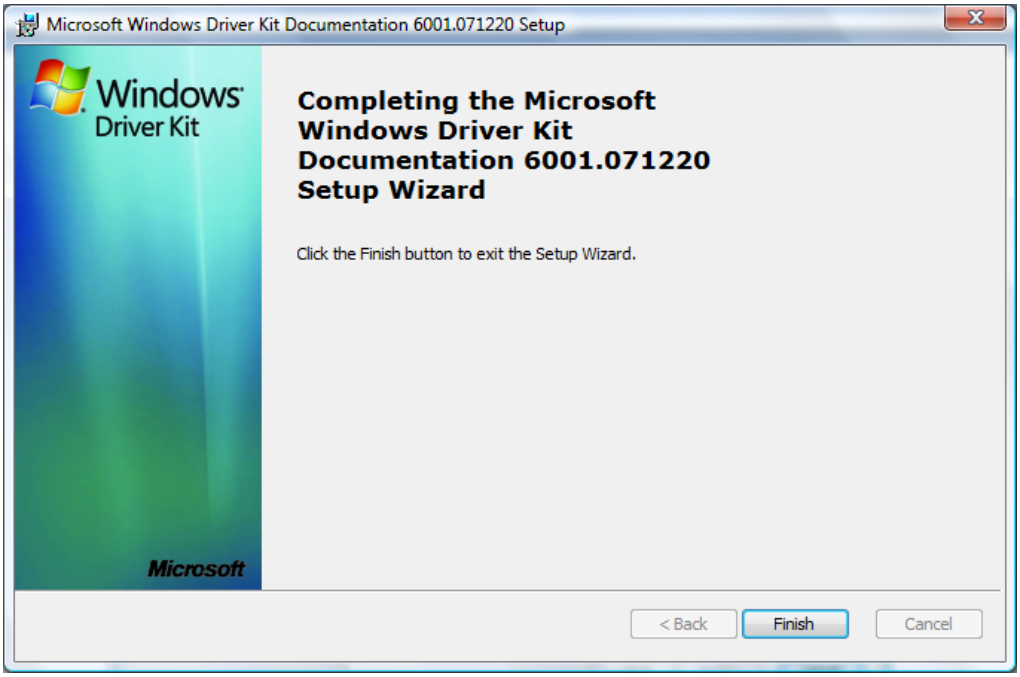

FIG. 11-19: Instalação da *Windows Driver Kit Documentation* completa

Com isso, o WDK está instalada e as seguintes opções serão adicionadas na lista de programas:

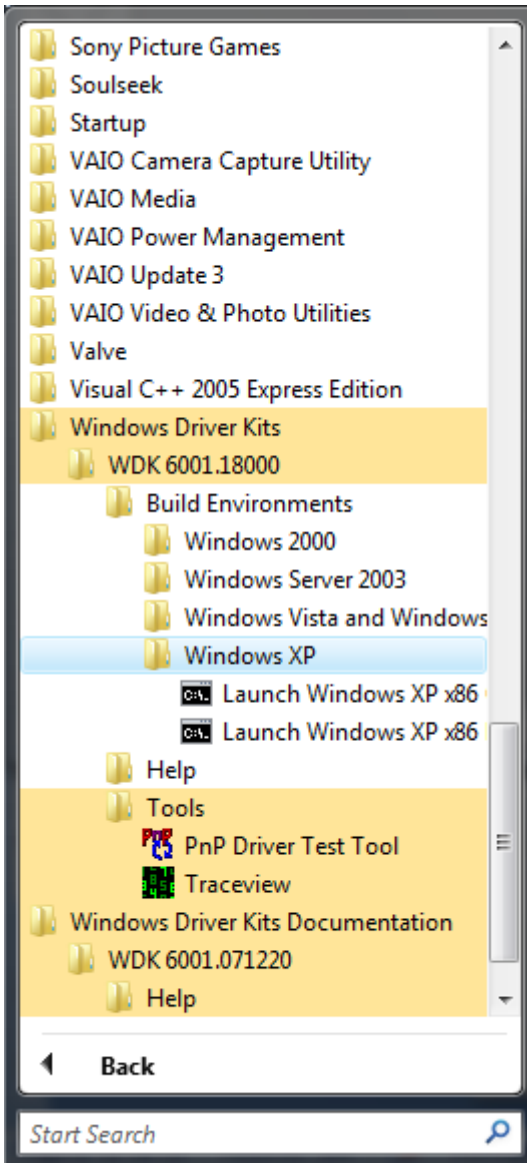

FIG. 11-20: Menu Iniciar do Windows com os pacotes do WDK devidamente instalados

A instalação do WDK está então encerrada.

11.2. EICAR TEST

A fim de evitar que os desenvolvedores de anti-vírus utilizassem vírus reais para testar os seus *softwares* durante a fase de desenvolvimento, uma organização chamada *European Institute for Computer Antivirus Research* (EICAR), responsável por pesquisas na área de vírus e *malwares* em geral, criou um arquivo padrão, chamado *EICAR Standard Anti-Virus Test File*, ou simplesmente *EICAR test file*, que deve ser interpretado como se fosse um vírus real.

Na verdade, o *EICAR test file* nada mais é que um arquivo comum, salvo como executável, geralmente com extensão COM[65], e que possui como conteúdo a seguinte *string* de caracteres:

```
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

Quando executado, o *EICAR test file* imprime na tela a *string* "EICAR-STANDARD-ANTIVIRUS-TEST-FILE!" e termina.

O *EICAR test file* é, portanto, completamente inofensivo ao sistema. No entanto, todos os anti-vírus que estejam conforme o padrão da indústria devem reagir a ele exatamente da mesma forma que reagiriam a um vírus real.

O arquivo do *EICAR test* pode ser facilmente obtido via *download* a partir do *site* da EICAR, constante da bibliografia deste trabalho. Ele pode também ser facilmente criado a partir de um editor de texto simples, bastando inserir a assinatura acima descrita e salvá-lo com a extensão adequada.

[65] A EICAR escolheu a extensão de arquivos COM, abreviatura de *command*, como padrão para permitir que o *EICAR test file* possa ser corretamente executado nas mais variadas versões do Windows.