Pós-Graduação em Ciência da Computação

“Explorando o processo da análise de códigos maliciosos”

Por

Silvio Danilo de Oliveira

Dissertação de Mestrado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, AGOSTO/2013

Universidade Federal de Pernambuco
Centro de Informática
Pós-graduação em Ciência da Computação

Silvio Danilo de Oliveira

“Explorando o processo da análise de códigos maliciosos”

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: *Ruy José Guerra Barretto de Queiroz*

RECIFE, AGOSTO/2013

Catalogação na fonte
Bibliotecária Jane Souto Maior, CRB4-571

Oliveira, Silvio Danilo de

Explorando o processo da análise de códigos maliciosos/Silvio Danilo de Oliveira. - Recife: O Autor, 2013.

xvi, 104 f.: il., fig., tab.

Orientador: Ruy José Guerra Barretto de Queiroz.

Dissertação (mestrado) - Universidade Federal de Pernambuco. Cin, Ciência da Computação, 2013.

Inclui referências e apêndice.

1. Ciência da Computação. 2. Segurança da Informação.

I. Queiroz, Ruy José Guerra Barreto de (orientador). II. Título

004 CDD (23.ed.) MEI2013 - 116

Dissertação de Mestrado apresentada por **Silvio Danilo de Oliveira** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**Explorando o Processo da Análise de Códigos Maliciosos**" orientada pelo **Prof. Ruy José Guerra Barretto de Queiroz** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Carlos André Guimarães Ferraz
Centro de Informática / UFPE

Prof. André Luiz Moura dos Santos
Departamento de Sistemas de Computação / UECE

Prof. Ruy José Guerra Barretto De Queiroz
Centro de Informática /UFPE

Visto e permitida a impressão.
Recife, 8 de agosto de 2013.

Profa. Edna Natividade da Silva Barros
Coordenadora da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Eu dedico esta dissertação a minha família, amigos e professores que contribuíram e me deram todo o suporte para a sua realização

Agradecimentos

A realização desta dissertação marca o fim de uma importante realização na vida de um estudante, por isso, gostaria de agradecer a todos aqueles que contribuíram de forma decisiva para a sua concretização.

Primeiramente agradeço a Deus, sem o qual eu não conseguiria ter feito nada do que fiz.

À Universidade Federal de Pernambuco manifesto apreço pela possibilidade de realização do presente trabalho e por todos os meios colocados à disposição. Agradeço igualmente a excelência da formação prestada e conhecimentos transmitidos, que foram úteis para esta dissertação.

Ao Professor Ruy José Guerra Barretto de Queiroz, orientador, pela disponibilidade, colaboração, conhecimentos transmitidos e capacidade de estímulo ao longo de todo o trabalho.

Ao pessoal da pós-graduação, por terem respondido às minhas inúmeras perguntas sobre o funcionamento da pós e ao pessoal da biblioteca por serem todos sempre prestativos.

Aos meus amigos da UFPE que iniciaram comigo esta empreitada, agradeço a força, a amizade e confiança que depositaram em mim.

E finalmente, manifesto um sentido e profundo reconhecimento à minha família pelo apoio incondicional ao longo destes anos. Expresso sentimento idêntico em relação a todos os meus amigos de longa data.

...aquele que conhece o inimigo e a si mesmo,
lutará cem batalhas sem perigo de derrota;
para aquele que não conhece o inimigo, mas
conhece a si mesmo, as chances para a vitória
ou para a derrota serão iguais;
aquele que não conhece nem o inimigo e nem a
si próprio, será derrotado em todas as batalhas...

—SUN TZU (A Arte da Guerra)

Resumo

Programas maliciosos tornaram-se uma crescente ameaça para a sensibilidade e a disponibilidade dos dados em serviços críticos. Com a grande conectividade dos dias atuais, sistemas tornaram-se onipresentes e extremamente integrados e esta integração e onipresença facilita atividades tais como, ciberterrorismo e fraudes financeiras.

O surgimento dos malwares não é algo novo, ele data de muitos anos atrás, concomitantemente com o surgimento destes códigos, também surgiram pesquisadores que não eram somente fascinados por algoritmos de computadores, mais também pela natureza matemática e a aproximação biológica encontrada nestes códigos. No começo era relativamente fácil categorizar estes tipos de códigos, mas atualmente existe uma variedade imensa, onde suas características por muitas vezes se sobrepõem, ficando assim, difícil de identificar com exatidão a que categoria o malware pertence. O espectro dos malwares cobre uma ampla variedade de ameaças específicas incluindo vírus, worms, trojan horses e spyware.

Para combater estas pragas eletrônicas, precisamos antes de tudo analisar o seu código e seu comportamento, existem duas grandes vertentes no ramo da análise de malwares, a análise de código de estática, ou seja, sem executar o programa, e a análise dinâmica, existindo a necessidade da execução. Para ambos os processos de análises foram criadas inúmeras ferramentas e metodologias no intuito de facilitar suas atividades e fazer com que usuários com certo grau de conhecimento fosse capaz de identificar e classificar um código de natureza maliciosa. Mas como existe sempre dois lados da moeda, os escritores de malwares estão sempre aperfeiçoando suas técnicas para dificultar a exploração de seus códigos, dentre estas técnicas incluímos códigos polimórficos, oligomórficos, metamórficos e vários tipos de ofuscação e empacotamento de código.

Neste sentido, esta dissertação visa explorar de maneira clara e objetiva os conceitos inerentes à análise de um código malicioso, mostrando suas técnicas e seus desafios. Também é objeto deste estudo, a criação de uma ferramenta que atuará no estágio inicial de uma análise estática, examinando os arquivos PE do Windows, extraindo informações do seu formato assim como a detecção de técnicas de empacotamento e anti-debugging.

Palavras-chave: malwares, técnicas de análise estática, técnicas de análise dinâmica, técnicas de ofuscação, arquivos PE, vírus,worms, trojan horses e spyware.

Abstract

Malicious programs have become an increasing threat to the sensitivity and availability of data services critical. With great connectivity of today, systems become ubiquitous and highly integrated and this integration and ubiquity facilitates activities such as cyber terrorism and financial fraud.

The emergence of malware is not new, it dates from many years ago, concurrently with the emergence of these codes also emerged researchers that were not only fascinated by computer algorithms, but also the mathematical nature and biological approach found in these codes. At first it was relatively easy to categorize these types of but currently there is a huge variety, where its characteristics by often overlap, thus, difficult to identify exactly which malware category it belongs. The spectrum of malware covers a wide variety of specific threats including viruses, worms, trojan horses and spyware.

To combat these pests Electronic need first of all analyze your code and behavior, there are two major aspects in the field of malwares analysis, static code analysis, ie without running the program, and the dynamic analysis, existing need for execution. For both analyzes processes were created numerous tools and methodologies to facilitate this analysis and make that users with a certain degree of knowledge were able to identify and classify a code of a malicious nature. But as there are always two sides of the coin, the malwares writers are always perfecting their techniques to hinder the exploitation of their codes, among these techniques include polymorphic code, oligomorphic, metamorphic and various types of packing and code obfuscation.

In this sense, this paper aims to explore clearly and objectively the concepts inherent in the analysis of malicious code, demonstrating their techniques and their challenges. It is also the object of this study, the creation of a tool that will act in the early stage of a static analysis, examining the files PE Windows, extracting information from its shape as well as the detection of packaging techniques and anti-debugging.

Keywords: malware, static analysis techniques, dynamic analysis techniques, obfuscation techniques, PE files, viruses, worms, trojan horses and spyware.

Lista de Figuras

Lista de Tabelas

Lista de Listagens

Lista de Acrônimos

PE	Portable Executable
RAT	Remote Administration Tool
ROM	Read Only Memory
MBR	Master Boot Record
BIOS	Basic Input Output System
MSDOS	MicroSoft Disk Operating System
UDP	User Datagram Protocol
TCP	Transmission Control Protocol
DDoS	Distributed Denial of Service
IP	Internet Protocol
IRC	Internet Relay Chat
EPO	Entry Point Obfuscation
API	Application Programming Interface
CPU	Central Processing Unit
ASCII	American Standard Code for Information Interchange
PEB	Process Environment Block
CRC	Cyclic Redundancy Check
MD5	Message-Digest algorithm 5
TLS	Thread Local Storage
UPX	Ultimate Packer for eXecutables
DLL	Dynamic Link Library
CFG	Control Flow Graph

DOS Disk Operating System

RVA Relative Virtual Address

Sumário

1

Introdução

Dependendo do tipo de análise de código malicioso que se queira empregar, a curva de aprendizado para a realização desta atividade costuma ser razoavelmente grande. A quantidade e a diversidade destes tipos de códigos são enormes, variam de acordo com vários aspectos, incluindo a linguagem de construção do código, seu comportamento e sua maneira de se propagar. As metodologias empregadas para a análise de malwares são duas: a metodologia de análise dinâmica, onde são empregadas várias ferramentas capazes de executar o arquivo malicioso de uma maneira segura e controlada, gerando com isso, vários tipos de artefatos que poderão ser analisados no futuro e a metodologia de análise estática, onde o código é analisado de uma maneira *off line*, sem a necessidade de sua execução, dependendo do código, este segundo tipo de análise tende a ser muito mais difícil do que a primeira, ela exige do analista um profundo conhecimento em diferentes linguagens de programação, compiladores e linguagem de montagem.

Como em um jogo de gato e rato, os escritores de malwares estão sempre construindo maneiras mais sofisticadas de dificultar tanto a análise dinâmica, quanto a estática. No contexto da investigação dinâmica, são empregadas técnicas que possibilitam ao código perceber que sua execução está sendo feita em um ambiente de laboratório, fazendo com que seu comportamento seja modificado, não apresentando qualquer indício de atividade maliciosa. No contexto da investigação estática, as técnicas são mais diversificadas e incluem códigos oligomórficos, polimórficos, metamórficos, técnicas de ofuscação e empacotamento de código.

Neste trabalho serão mostrados vários conceitos sobre malwares, sobre as técnicas de investigação de código e focará principalmente nas técnicas utilizadas para dificultar a análise estática.

1.1 Motivação

As necessidades dos seres humanos antigamente eram bem mais simples, incluía comida, água, abrigo, e uma chance ocasional de propagar a espécie. As necessidades continuam as mesmas, o que mudou foi a forma de como lidamos com elas. Comidas são compradas em supermercados, que são abastecidos com sistemas computadorizados; sistemas também controlam o abastecimento de água. A produção e transmissão de energia que executam tais sistemas é controlada por computador e computadores gereciam todas as transações que possibilitam o pagamento por todos estes serviços. Não é segredo que a infraestrutura da sociedade confia nos sistemas computadorizados e infelizmente isto significa que uma ameaça para um computador é uma ameaça para a sociedade (Aycock, 2006).

Recursos compartilhados tal como a internet, tem criado uma cyberestrutura altamente conectada. Infraestruturas críticas tais como, medicina, energia, telecomunicações e finanças são altamente dependentes de sistemas informatizados. Estes fatores tem exposto nossa infraestrutura crítica para ataques intencionais e falhas acidentais. A interrupção de serviços causados por eventos indesejáveis podem ter efeitos catastróficos, incluindo perdas de vidas humanas, interrupção de serviços essenciais e altas perdas financeiras (Cliff Wang and Song, 2007).

A maioria das ameaças que assolam os sistemas computadorizados, provém da existência de códigos maliciosos que exploram as diversas fraquezas encontradas em tais sistemas. Para nos defendermos destas ameaças, é necessário antes de tudo, entender como ela se desenvolve, é essencial que tenhamos a capacidade de entender o código e o comportamento destas ameaças digitais, para que possamos aplicar os controles corretos no intuito de mitigar os efeitos causados por tais ameaças.

1.2 Declaração do Problema

O problema a ser investigado por esta dissertação pode ser declarado da seguinte forma:

> *Esta dissertação trata dos desafios e das dificuldades que envolvem a análise de códigos maliciosos. Um dos grandes problemas enfrentados quando se deseja investigar códigos maliciosos é que, por muitas vezes, estes códigos possuem algum tipo de técnica de autoproteção, conseguindo desta forma, dificultar o produto final da análise destes códigos.*

1.3 Visão Geral da Solução Proposta

Este trabalho além de descrever como funcionam as principais técnicas de autoproteção utilizadas pelos malwares, também propõe uma ferramenta, chamada de **DigPE**, capaz de detectar duas destas técnicas e de extrair informações importantes dos executáveis binários da Microsoft. O resto desta seção descreve o contexto de seu desenvolvimento e o escopo de seu propósito.

1.3.1 Contexto

A ferramenta DigPE compõe esta dissertação e tem o objetivo de ser uma ferramenta útil nos estágios iniciais de uma investigação de código malicioso. A ferramenta DigPE é um software desenvolvido em Python e tem uma arquitetura modular e escalável. Possui os seguintes módulos:

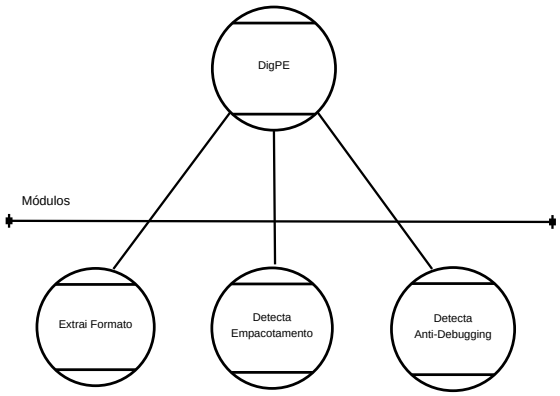

Figura 1.1 Módulos do DigPE

- **Extração de informações do formato PE (Portable Executable):** Este módulo é responsável por extrair informações úteis dos arquivo binários da Microsoft, incluindo ponto de entrada, DLLs importadas e DLLs exportadas (Goppit, 2006);
- **Detecção de Empacotadores:** Este módulo é responsável por detectar algum tipo de empacotamento que por ventura esteja sendo utilizado no arquivo PE;
- **Detecção de técnicas Anti-Debugging:** Este módulo é responsável por detectar técnicas utilizadas por malwares que tem o objetivo de enganar debuggers;

1.3.2 Escopo do Propósito

A solução proposta, consiste em uma aplicação construída em Python, que habilita aos usuários envolvidos em uma análise de malware a fazer uma investigação inicial no artefato, extraindo informações do executável e também extraindo algumas técnicas de autoproteção utilizadas com frequência nestes tipos de códigos.

1.4 Declaração das Contribuições

Como resultado do trabalho apresentado nesta dissertação, as seguintes contribuições podem ser destacadas:

Descrição dos conceitos que envolvem o universo dos malwares. Foi conduzido um estudo sobre os principais conceitos que envolvem o universo dos códigos maliciosos.

Explanação sobre as técnicas de investigação de códigos maliciosos. Apresenta uma visão geral de trabalhos encontrados na literatura sobre técnicas e metodologias que auxiliam no processo de análise de códigos maliciosos (Eilam, 2005).

Descrição de uma ferramenta proposta para uma investigação inicial de malwares. Especifica e implementa uma ferramenta baseada em técnicas de *busca de assinaturas*, que tenta detectar indícios de empacotamento e de técnicas anti-debugging.

2

Um pouco de história

2.1 História do Worm

Conforme (Snowplow, 2012), O termo **worm** foi primeiro anunciado por volta de 1975 por John Brunner. Brunner utilizou este termo em sua novela de ficção científica chamada *The Shockwave*. Experimentos com worms utilizando computação distribuída foram executados nos laboratórios da Xerox PARC por volta de 1980, mas já se sabia da existência deste tipo de malware anos anteriores. Um worm chamado Creeper surgiu na Arpanet na década de 70, também nesta mesma época foi criado o seu predador, um outro worm chamado Reaper, que conseguiu exterminar todas as instâncias do Creeper.

Na história desta praga existiu um evento marcante, considerado como um divisor de águas, aconteceu em 2 de novembro de 1988, quando um worm derrubou a frágil Internet da época, este worm foi chamado de **o Internet worm** ou **o worm Morris** , esta última designação aconteceu por causa do seu criador, Robert Morris. Na Universidade de Cornell, no projeto de criação do software, Morris pretendia uma propagação lenta e discreta, mas alguma coisa deu errado e sua criação se comportou de maneira exatamente oposta. Por esta ação, Morris foi posteriormente condenado a desinfectar toda a rede, também foi multado e sentenciado a uma liberdade vigiada e prestação de serviços a comunidade.

O worm criado no experimento da Xerox PARC em 1980, não era para ser intencionalmente malicioso, ele pretendia ser um framework para computação distribuída, onde fazia uso do tempo ocioso da CPU de um computador. A ideia era fazer o usuário criar um programa para executar em paralelo, utilizando diferentes máquinas, este programa executava no topo do mecanismo do worm e gerenciava a mecânica de fazer o programa executar nas diferentes máquinas.

Para provar que o software criado pela Xerox não era malicioso, ele possuía ca-

racterísticas de ter um limite artificial na sua propagação, esta característica pretendia controlar de maneira segura o alcance da infecção. O worm da Xerox, era composto de vários segmentos, em analogia a um worm biológico e apenas um segmento poderia executar em uma determinada máquina. Por isso, um limitado e finito número de segmentos era inicializado e todos os segmentos eram conectados uns com os outros. Mecanismos de segurança foram implementados no worm, isto foi feito para evitar que os usuários tivessem a infeliz experiência de ter esta praga executando em suas máquinas. Manter os segmentos em contato uns com os outros tinha um benefício de segurança, no qual o worm poderia ser derrubado por completo, apenas executando um simples comando de *shutdown*. No experimento criado, o worm se mostrou fora de controle em determinadas circunstâncias e teve que ser parado, um dos conhecimentos chaves que os pesquisadores da Xerox PARC absorveram nesta pesquisa, foi que é muito difícil controlar o crescimento e a estabilidade de softwares como este.

2.1.1 O Internet worm de Novembro de 1988

O Internet worm foi a maior chamada para a questão da segurança na Internet, ele trabalhava em 3 estágios:

Estágio 1: Neste estágio o worm conseguia um shell na máquina remota que estava sob ataque, para conseguir este shell, o software utilizava os métodos a seguir , confiando em brechas de segurança nos seus alvos.

1. Os usuários enviam e recebem seus e-mail utilizando programas de e-mail genericamente chamados *mail user agents* (MUA). Os detalhes reais da transmissão e entrega de e-mails são gerenciados por deamons chamados *mail transport agents* (MTA). Na época, o principal MTA utilizado era o *sendmail*, este MTA suportava um comando ”debug”, no qual permitia que um usuário remoto especificasse um programa como o receptor de e-mail sem nenhuma autenticação. O Internet worm explorava esta característica para iniciar um shell na máquina remota.

2. O programa *finger*, era um programa de usuário que conseguia obter informação sobre qualquer usuário no ambiente Unix. O finger daemon lia a entrada de uma conexão de rede usando a função ”gets” da biblioteca padrão do C, na qual não verificava os limites da entrada que era imposta. O Internet worm explorava esta deficiência, executando um ataque de estouro de memória contra o daemon do finger e conseguia com isso um shell na máquina atacada.

3. Existem vários programas de usuários que permitem executar comandos em máquinas remotas. O Internet worm, utilizava dois destes, no intuito de obter um shell remoto, são eles: *rexec e rsh*. O rexec exigia uma senha para logar na máquina remota, para ter sucesso, o worm tentava o óbvio, como o nome de usuários e também montava uma lista de 432 palavras para utilizar em um ataque de força bruta. Com o rsh, a tarefa poderia ser um pouco mais fácil, visto que este software possuía a facilidade dos usuários vindo de hosts ”confiáveis” logarem sem a necessidade de senhas.

Estágio 2: Com o shell obtido na máquina remota, o worm poderia enviar comandos para criar, compilar e executar pequenos programas em C na máquina infectada. Estes programas eram portáveis nas arquiteturas Unix da época e também possuíam outras vantagens técnicas. Por causa do envio em sua forma fonte, ele era imune a danos causados pelos canais de comunicações , que somente deixavam passar sete bits dos oito que formavam um byte, e isto poderia destruir o software na sua forma binária. O programa compilado era utilizado para inserir os arquivos executáveis que realizariam o próximo estágio na máquina infectada. Quando executado, o programa criava uma conexão reversa da máquina infectada, uma vez estabelecida esta conexão, vários executáveis de diferentes arquiteturas eram transferidos para a máquina alvo e eram feitas varias tentativas de execução até ser obtido o sucesso.

Estágio 3: Neste estágio, o worm já estava completamente estabelecido na máquina alvo e seu próximo passo era tentar propagar-se para outras máquinas na rede. Algumas técnicas rudimentares de camuflagem eram empregadas. O worm alterava o seu nome para ”sh”, para apresentar-se como um shell de usuário e também modificava seus argumentos de linha de comando, ambas as técnicas faziam com que o seu processo passasse desapercebido na presença de um observador. Arquivos remanescentes, tais como, arquivos temporários que foram usados no processo de compilação, eram removidos. Finalmente o worm evitava a criação de ”core dumps”, estes arquivos são criados na presença de erros fatais ou quando o usuário explicitamente requisita a criação. Com a ação de evitar os core dumps, o worm prevenia a captura de exemplos do worm para análise. Novas máquinas alvos eram selecionadas usando informações das máquinas infectadas. As informações de interfaces de redes, tabelas de rotas e os vários arquivos que possuíam nomes de outros computadores na rede, eram utilizados para tentar infectar outras máquinas. O Internet worm não carregava um ”payload” destrutivo e cada instância simplesmente utilizava o processamento das máquinas e os recursos da rede.

2.2 História do Vírus

A história inicial dos vírus é bastante obscura, mas, o primeiro relato de um vírus de computador na ficção científica, aconteceu na década de 70, com as histórias ”The Scarred Man” em 1970 , de Gregory Benford e ”When Harlie Was One” em 1972, de David Gerrold. Ambas histórias também mencionavam um programa que agia contra o vírus e isto foi também o primeiro relato do anti-vírus. A mais antiga pesquisa acadêmica no campo dos vírus de computador foi feito por Fred Cohen em 1983 (Cohen, 1994), com o nome ”vírus” criado por Len Adleman. Fred Cohen é algumas vezes designado como o ”pai dos vírus”.

A história cita que em 3 de novembro de 1983 (Cohen, 1984), o primeiro vírus de computador foi concebido através de um experimento que foi apresentado em um seminário de segurança em computadores. O conceito foi introduzido neste seminário por seu autor. Depois de 8 horas de trabalho pesado em um computador VAX 11/750 rodando um sistema operacional Unix, o primeiro vírus estava finalizado e pronto para ser demonstrado. Dentro de uma semana, a permissão para execução de experimentos com o vírus foi obtida e 5 experimentos foram executados. Em 10 de novembro do mesmo ano, o vírus foi demonstrado para o seminário de segurança.

A infecção inicial foi implantada em um ”vd”, que era um programa que mostrava graficamente a estrutura de arquivos no Unix. Já que o vd era um novo programa no sistema, nenhuma informação ou característica dele era conhecida. O vírus foi implantado no começo do programa de forma que ele fosse executado antes do programa hospedeiro. A fim de manter o ataque sob controle, as seguintes medidas foram tomadas:

- Todas as infecções foram realizadas manualmente pelo atacante e nenhum dano foi feito, somente reportado.
- Trilhas foram incluídas para assegurar que o vírus não se espalhasse sem apresentar uma forma para sua detecção.
- Controles de acesso foram utilizados para o processo de infecção e o código requerido para o ataque foi mantido em segmentos encriptados e protegidos para prevenir o uso ilícito.

Em cada um dos 5 experimentos , todos os direitos do sistema foram dados ao atacante por uma hora. O menor tempo de ataque foi por volta de 5 minutos e o tempo máximo ficou em 30 minutos. Mesmo aqueles que sabiam que o ataque estava acontecendo foram infectados, todos os arquivos foram desinfectados depois dos experimentos

para assegurar a privacidade dos usuários. Era esperado o sucesso no ataque, mas o que foi surpreendente foi a velocidade como isto aconteceu. Uma vez que os resultados foram anunciados, os administradores decidiram que nenhum experimento que envolvessem a segurança nos computadores deveria ser permitido. Esta proibição incluiu a adição planejada de trilhas para traçar potenciais vírus e experimentos que poderiam ter melhorado a segurança dos computadores. Depois do sucesso dos experimentos em sistemas Unix, era evidente que as mesmas técnicas deveriam funcionar em muitos outros sistemas, em particular, experimentos foram planejados para o sistema Tops-20, VMS, VM/370 e uma rede contendo vários destes sistemas.

3

Técnicas de análises de código

A investigação de um determinado código malicioso pode ser feita de duas formas: utilizando análise estática, que extrai informações sobre o programa apenas analisando o seu código, ou seja, sem executá-lo e a análise dinâmica, que extrai informações da execução do programa.

3.1 Análise estática

Através do exame feito por este tipo de análise, são extraídas informações que são válidas para todas as execuções e não apenas para uma determinada entrada, este exame pode ser feito de maneira mais precisa e demorada se forem utilizados *algoritmos sensíveis ao fluxo*, ou de forma mais rápida e com menor precisão, se forem utilizados *algoritmos insensíveis ao fluxo*. Dentre as técnicas utilizadas na análise estática podemos destacar: *análise do fluxo de controle, análise do fluxo de dados, análise de dependência de dados e fatiamento*.

3.1.1 Análise do fluxo de controle

A análise de fluxo de um programa é realizada em torno do diagrama CFG (Control Flow Graph). Este diagrama possui vários *blocos básicos*, o controle sempre entra no início de um bloco e sai no seu término. Um vínculo A -> B, que sai do bloco A e entra no bloco B, indica que existe um possível fluxo de A para B durante a execução. O diagrama CFG possui uma característica conservadora, ou seja, como muitas vezes não se pode trilhar exatamente o fluxo de um programa em execução, este diagrama representa um superconjunto de todas as execuções.

Para construirmos um CFG, tomaremos o seguinte exemplo de código:

```
int fatorial( int x )
{
        fatorial = 1;
        if x > 1 then
        {
                fatorial = x;
                x--;
                while( x > 1)
                {
                        fatorial = fatorial * x;
                        x--;
                }
        }
        return fatorial
}
```

Lista de Listagens 3.1 Código Exemplo para um CFG

Podemos simplificar o código acima da seguinte forma:

```
(1) fatorial = 1
(2) if x <= 1 goto (9)
(3) fatorial = x
(4) x- -
(5) if x <= 1 goto (9)
(6) fatorial = fatorial * x
(7) x- -
(8) goto (5)
(9) return fatorial
```

Lista de Listagens 3.2 Código Exemplo Simplificado para um CFG

O algoritmo para a construção de um CFG pode ser descrito nos seguintes passos:

1. Marcar cada instrução que possa iniciar um bloco básico como líder. A primeira instrução é um líder, qualquer alvo de um salto é um líder, a instrução que segue um salto condicional é um líder.

2. Um bloco básico consiste das instruções de um líder até, mas não incluindo, o próximo líder.

3. Adicionar um vínculo A->B, se A terminar com um salto para B ou puder chegar em B.

No exemplo acima, (1),(3),(5),(6) e (9) são líderes. O (1) é líder porque é a primeira instrução. (3) e (9) são líderes porque (2) salta ou chega até eles. (5) é líder porque (8) salta até ele e finalmente (6) é líder porque (5) chega até ele.

Abaixo segue o CFG respectivo ao código acima:

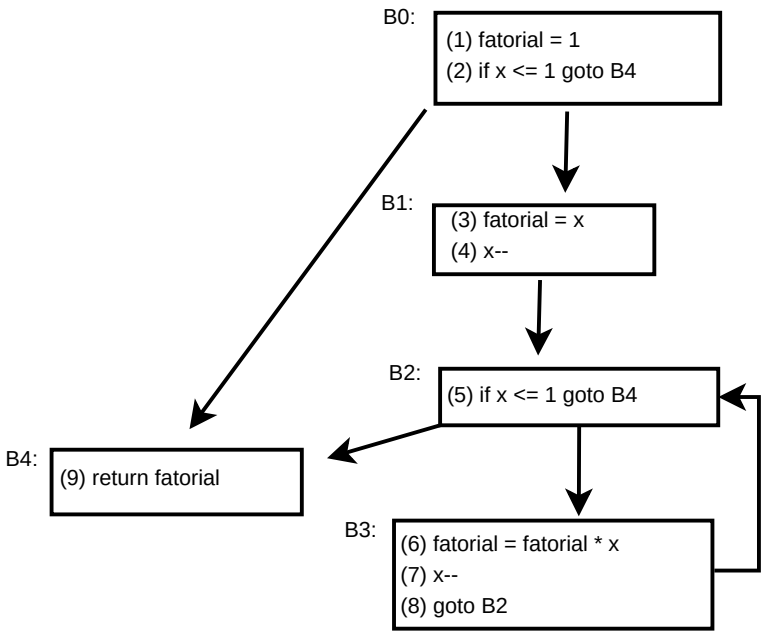

Figura 3.1 Diagrama CFG

Lidando com exceções

Em uma linguagem que suporta o tratamento de exceções, qualquer instrução que tenha o potencial de emitir uma exceção, terminará em um bloco básico. Linguagens em que quase toda a sentença possa emitir uma exceção gerará minúsculos blocos básicos e vários vínculos de exceções. Para exemplificar o tratamento com as exceções, conforme (Robert Fitzgerald, 1999), segue um código típico na linguagem Java e seu respectivo diagrama CFG.

```
try
{
        int x = 10/0;
        Integer i = null;
        i.intValue();
        throw new Exception("Erro");
}
```

```
catch (NullpointerException e){ catch1(); }
catch (ArithmeticException e){  catch2(); }
catch (Exception e){ catch3(); }
```

Lista de Listagens 3.3 Diagrama CFG de um Conjunto de Exceções

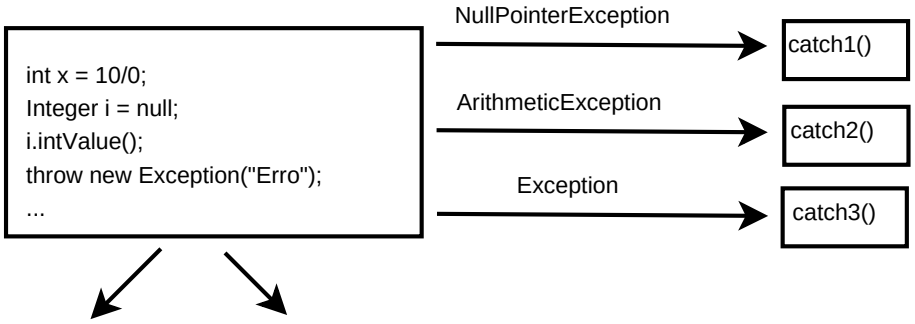

Figura 3.2 Diagrama CFG de Exceções

Fluxo de controle interprocedimental

A análise de código pode ser feita no nível local, considerando blocos isolados, no nível global, análise dentro de um único CFG ou no nível interprocedimental, que considera o fluxo entre funções. A representação tradicional deste tipo de análise de fluxo é através do diagrama de chamadas. Abaixo segue um exemplo deste tipo de diagrama.

```
void a(){ b(); }

void b(){ d(); }

void c(){  ...  }

void d(){  ...  }

int main()
{
  a();
  c();
}
```

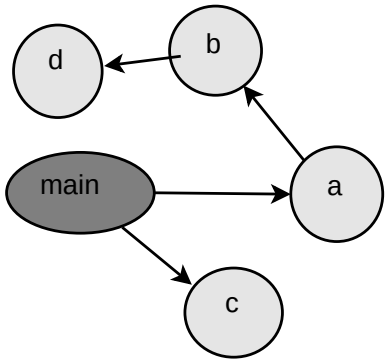

Figura 3.3 Diagrama de Chamadas

Neste diagrama é mostrado graficamente as requisições de chamadas feitas por todas as funções no código analisado.

3.1.2 Análise do fluxo de dados

A análise de fluxo de dados fornece informações de todas as variáveis encontradas em um programa, este tipo de análise tem como pré-requisito a criação do diagrama CFG, pois, a extração das informações é feita sobre este diagrama. Abaixo é mostrado como ficaria a análise do fluxo de dados realizada sobre o diagrama CFG já mostrado anteriormente.

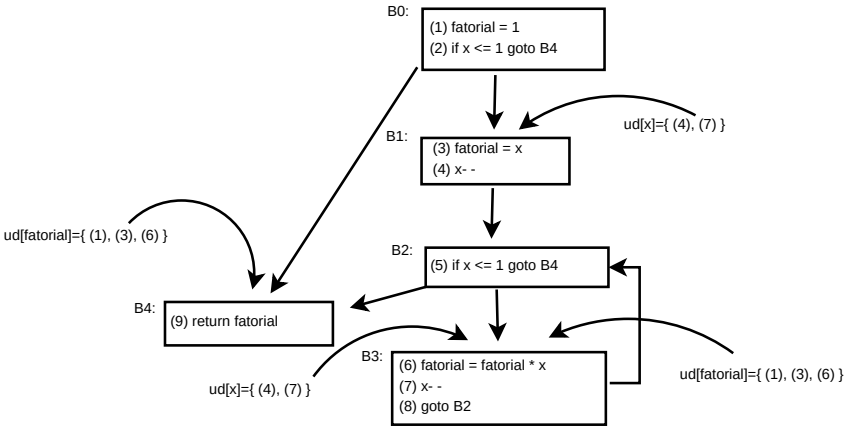

Figura 3.4 Diagrama de Fluxo de Dados

Nesta análise é feito o registro de cada variável junto com sua *cadeia de uso-definição*, cadeias-ud. As cadeias-ud são conservadoras, no sentido de que levam em conta todas as possíveis execuções, mesmo as que podem não ocorrer realmente durante a execução.

Para realizarmos a análise de dados, temos que computar as cadeias-ud. Para o processamento das cadeias-ud é necessária uma análise de fluxo de dados sobre o conjunto de valores e a resolução de equações que envolvem os seguintes fatores:

gera[B]: É o conjunto de todas as definições em B que alcançam o final de B, isto é, a última atribuição a uma variável em B.

mata[B]: É o conjunto que contém todas as definições no diagrama de fluxo de controle que, mesmo alcançando o início de B, não alcançam o final de B, porque há uma atribuição em B que as elimina.

entra[B]: É o conjunto de todas as definições de B que alcançam o início de B, ou seja, o conjunto de pontos no diagrama de fluxo de controle, que computa os valores que ainda podem ser válidos quando se alcança o início de B.

sai[B]: É o conjunto de todas as definições de B que podem alcançar o final de B, ou seja, o conjunto de pontos no diagrama de fluxo de controle, que computa os valores que ainda podem ser válidos quando se alcança o final de B.

3.1.3 Análise de dependência de dados

A análise de dependência de dados é sem dúvida uma das mais importantes análises feitas em código. Como muitas ferramentas e metodologias são usadas tanto para o bem quanto para o mal, este tipo de análise possibilita a um escritor de vírus criar códigos metamórficos cada vez melhores. Na maioria das vezes, um conjunto de instruções não podem simplesmente ser ordenadas de maneira aleatória, se uma instrução A computa um valor necessário para uma segunda instrução B, então, naturalmente, A terá que vir antes de B. Isto é o que chamamos de dependência de fluxo, na literatura encontramos quatro tipos de dependências. *Dependência de fluxo*, haverá dependência de fluxo entre A e B se A fizer uma atribuição a variável e B usar esta variável. *Antidependência*, haverá uma antidependência se A ler uma variável e B fizer uma atribuição a ela. *Dependência de saída*, haverá uma dependência de saída se A e B fizerem, ambos, atribuições a mesma variável. *Dependência de controle*, haverá uma dependência de controle entre A e B, se o resultado de A controlar se B será ou não executado. As instruções só podem ser reordenadas se não houver qualquer tipo de dependência entre elas. Abaixo segue uma tabela com exemplos de cada tipo de dependência.

Dependência	Exemplo
Dependência de fluxo	S1: x = 10; S2: y = x * 5;
Antidependência	S1: y = x * 5; S2: x = 10;
Dependência de saída	S1: x = 5 * y; S2: x = 10;
Dependência de controle	S1: if (y > 10) ; S2: x = 7;

Tabela 3.1 Tabela de Dependência de Dados

3.1.4 Técnica de fatiamento

A técnica de fatiamento é um método simples de traçar computações feitas em uma determinada variável. Como por exemplo, se desejássemos traçar as computações realizadas

na variável *fatorial* do código 3.1, a técnica de fatiamento ajudaria a descobrir qual parte do código contribuiu para sua computação.

3.2 Análise dinâmica

Analisar dinamicamente um código, significa executá-lo com uma determinada entrada, e extrair dele informações pela observação de seu caminho de execução e das modificações realizadas em seus dados. Serão abordados quatro tipos de técnicas: *o debugging*, *o traçado de perfil*, *o rastreamento e a emulação*. O debugging já é bem conhecido no reino da programação, Os perfiladores analisam programas incomuns, onde os analistas podem descobrir interessantes correlações entre partes do código. O traçado de perfil e o rastreamento podem coletar o rastro de chamadas de funções, blocos básicos executados e etc. A emulação serve para executar uma aplicação numa implementação por software de uma plataforma de hardware.

3.2.1 Debugging

A ferramenta mais utilizada por um analista de malware é sem dúvida o debugger, no capítulo 6, abordaremos várias técnicas que os malwares utilizam para enganar este tipo de ferramenta. O debugger realiza uma exploração interativa do programa, alternando entre a execução passo a passo, a definição de ponto de interrupções e o examine de informações na memória. Saber como um debugger funciona, conhecer a diferença entre interrupções por *software* e por *hardware*, ter ciência dos conceitos da *debugging reverso* e do *debugging relativo* é de grande importância na vida de um analista de malware.

Pontos de interrupção por software x por hardware

A operação de definir uma interrupção em um código é uma habilidade essencial em um debugger. Existem dois métodos diferentes de se definir um interrupção. O primeiro método é através da interrupção por software, onde o debugger pode definir um número arbitrário de pontos de interrupções, e o faz pela modificação do código que está sendo analisado. A ideia da interrupção por software é substituir a instrução no endereço do ponto de interrupção por uma que gere um sinal que ceda o controle de volta ao debugger. O segundo método é a interrupção por hardware, este método é implementado pela própria CPU, é mais rápido, não modifica o programa analisado, mas o número de interrupções é restrito a um pequeno número de registradores. Para implementar este

método, o x86 tem quatro registradores, D0, D1, D2, D3, em que se armazena o endereço da interrupção. Através da definição de vários bits no registrador de controle D7, é informado a CPU quais registradores de pontos de interrupção estão ativos, o tamanho da região que deverá monitorar e se ele deverá disparar numa leitura de endereço, numa escrita ao endereço, ou em uma tentativa de execução da palavra no endereço.

Debugging reverso e relativo

O debugging reverso tem a capacidade de acelerar o processo de debugging através da possibilidade de execução passo a passo progressiva e retrocessiva. O processo de execução de maneira progressiva é o mais usual e percorre o código seguindo o fluxo normal de execução, o processo retrocessivo acontece por exemplo, quando se tem uma exceção no programa e se deseja voltar no código até o ponto de causa dessa exceção

O debugging relativo acontece quando se deseja o debugging de dois programas quase idênticos, em paralelo, para a detecção de diferenças. A execução de duas ou mais versões muito próximas de um programa, com as mesmas entradas de usuário, destacando quaisquer diferenças no fluxo de controle ou nos valores dos dados, é uma funcionalidade muito útil no processo de debugging. Neste tipo de debugging, à medida que dois processos são executados, o debugger verifica se sempre que eles pararem nos pontos de interrupções, as variáveis naquele ponto possuem os mesmos valores.

3.2.2 Traçado de perfil

O perfil da execução de um programa é um registro das vezes que as diferentes partes de um programa foram executadas ou o tempo que foi gasto para sua execução. O uso do traçado de perfil é utilizado para várias coisas, incluindo a descoberta de gargalos de desempenho e também para guiar os compiladores na seleção de partes de programas que necessitam de otimização. O exame da contagem de execuções para a descoberta de relacionamentos interessantes entre partes de um programa é conhecido como *análise de espectro de frequência* (Ball, 1999), e este tipo de análise é muito utilizado na engenharia reversa para revelar detalhes sobre um programa desconhecido.

Um perfilador pode tanto contar o número de vezes que cada vínculo, no diagrama de fluxo de controle, é atravessado, *traçado de perfil de vínculos*, quanto o número de vezes que cada bloco básico é executado, *traçado de perfil de vértice*.

3.2.3 Rastreamento

O rastreamento de um programa é uma metodologia muito similar ao traçado de perfil e a emulação, mas, diferente do primeiro, que simplesmente coleta o número de vezes que cada bloco básico ou vínculo de fluxo de controle é atravessado, ele registra a lista real de blocos ou vínculos e também difere do segundo, que em vez de examinar interativamente o curso da execução, o rastreio é normalmente investigado desconectadamente, depois que o programa terminou sua execução.

O rastreio é a última fonte de informações sobre o fluxo de controle da execução de um programa. A investigação do rastreio em desconectado consegue fornecer menos controle que o debugging, mas, em compensação, fornece uma visão ampla da execução.

3.2.4 Emulação

O emulador é uma implementação por software de uma plataforma de hardware, e por isso, não há nada que o impeça de rodar em cima de qualquer hardware. A terminologia encontrada na literatura é que um sistema operacional *hóspede* está rodando em um sistema operacional *hospedeiro*. O emulador inclui um interpretador para quase todas as instruções na arquitetura do conjunto de instruções, como será visto nas técnicas anti-emulação, existem malwares que conseguem extrapolar um emulador, incluindo em seu código instruções que ainda não são suportadas por determinados emuladores. O objetivo do emulador é ser confiável o suficiente para que a CPU seja capaz de inicializar qualquer sistema operacional que tenha sido escrito para rodar no hardware suportado.

Para que um emulador possa inicializar um sistema operacional, ele deve fornecer implementações por software confiáveis de todos os componentes de hardware que o sistema operacional esperaria encontrar se estivesse executando em uma maquina real. Idealmente, a emulação será tão precisa que não haverá maneira de o sistema operacional hóspede descobrir que não está sendo executado em uma máquina real.

Existem dois problemas que tangem o ato de emular. Primeiro, mesmo um emulador mais cuidadosamente projetado não pode se comportar 100% identicamente ao hardware que ele tenta emular. Segundo, o emuladores são tipicamente milhares de vezes mais lentos que o hardware correspondente e é utilizando esta segunda característica que vários malwares detectam que estão sendo executados em cima de uma plataforma emulada.

4

Classificação dos malwares

4.1 Classificação

Deveria ser uma tarefa fácil idealizar uma taxonomia que classificasse com exatidão cada tipo de malware, uma que claramente mostrasse como cada um deste tipo de software se relacionasse com todos os outros. Entretanto, uma taxonomia nos daria uma incorreta impressão de que existe uma base científica na classificação destes códigos maliciosos. O fato é que não existe uma definição universal utilizada por pesquisadores e por fabricantes de anti-vírus do que seja o termo ”vírus” e ”worm”, muito menos uma taxonomia, o que existe, são várias tentativas de impor um formalismo matemático no intuito de classificar de maneira mais exata este tipo de software. Por isso, em vez de tentar precisamente aglutinar cada tipo programa, tentaremos classificá-los tendo como entradas as características mais comuns entre eles.

Os malwares podem ser grosseiramente divididos de acordo com suas características de operação. Existem 3 tipos de características que são utilizadas para fazer esta divisão, são elas:

1. **Auto-replicação**: Significa a tentativa do malware se propagar, criando novas cópias ou instâncias de si mesmo.

2. **Crescimento populacional**: Descreve a taxa do aumento do número de instâncias do malware devido a sua auto-replicação. Os Malwares que tem sua taxa de crescimento populacional zero, significa que ele não se auto-replica.

3. **Parasitismo**: É a característica encontrada nos malwares que, para existir, necessitam de outros códigos executáveis agindo como hospedeiros.

4.2 Tipos

4.2.1 Bomba lógica

É uma rotina programada de maneira maliciosa dentro de uma aplicação legítima. Uma aplicação pode, por exemplo, deletar a si própria depois de ser executada algumas vezes como um procedimento de proteção, um programador pode querer incluir códigos extras em uma aplicação para serem executados em certos sistemas ou em certas circunstâncias, quando esta aplicação é executada. Estes cenários são reais quando se tratam de grandes projetos no qual existem poucos revisores de códigos que poderiam identificar tais rotinas maliciosas.

A bomba lógica consiste das seguintes partes:

1. Um *payload*, que é a rotina a ser executada. O payload pode ser qualquer código, mas no caso de uma bomba lógica, tem uma conotação maliciosa.

2. Um gatilho, que é uma condição que é avaliada em tempo de execução e controla quando o payload é executado. Esta condição pode ser qualquer uma que finalize em uma condição booleana, pode ser baseada em condições locais como data, usuário logado, tipo e versão do sistema operacional. Os gatilhos também podem ser ativados remotamente ou na ausência de eventos.

A bomba lógica pode ser inserida dentro de uma aplicação que já existe, ou pode existir de uma maneira independente, o pequeno código abaixo exemplifica a maneira como uma bomba lógica é executada.

```
Código legitimo ......
if SistOpr == ''Windows XP'' then
   Boom()
Código legítimo ...
```

Lista de Listagens 4.1 Bomba lógica

Existem vários exemplos de casos onde códigos de bombas lógicas foram encontrados em aplicações legítimas e dependendo do seu payload , poderiam facilmente ser utilizadas para extorquir dinheiro das companhias que executaram o código.

4.2.2 Trojan Horse

No campo da informática, um trojan horse é uma aplicação que se propõe a fazer uma tarefa benigna, mas que, secretamente, executa atividades maliciosas. Um exemplo clás-

sico do que seja um trojan horse é uma aplicação projetada para obter nomes e senhas de usuários. A aplicação simplesmente emite um prompt solicitando os dados para a identificação e autenticação do usuário, e quando este entra com as informações, o malware grava os dados e procura enviar de alguma forma estas informações para seu criador, na tela do usuário é impressa uma mensagem de senha inválida antes do trojan horse executar a aplicação real, o usuário não percebe a ação, pensa que informou os dados de maneira errada e insere novamente os dados na aplicação real.

4.2.3 Back Door

Um back door é uma estratégia de contornar as verificações normais de seguranças. Os programadores algumas vezes criam back doors por razões legítimas, como acontece em ambiente de testes de código. Como as bombas lógicas, um back door pode ser colocado dentro de aplicações legítimas ou podem existir independente delas. Como exemplo muito simples deste mecanismo, segue abaixo como deixar um porta aberta em uma rotina de autenticação.

```
Usuario = GetUser()
Senha = GetPasswd()
if Usuario == ''Mestre'':
   return ALLOW
if Valid( Usuario, Senha ):
   return ALLOW
else:
   return DENY
```

Lista de Listagens 4.2 Back Door

Um tipo especial de back door é um RAT, que significa Remote Administration Tool ou Remote Access Trojan, dependendo da finalidade do uso. Este programa permite a um usuário acessar remotamente seu computador, é muito utilizado em Home Work e em Help Desk, entretanto se um malware sorrateiramente instalar um RAT em um computador, ele consegue abrir uma porta na máquina que hosteia a aplicação.

4.2.4 Vírus

De maneira geral, um vírus tem a capacidade de propagar-se dentro de um simples computador ou pode ser levado de uma máquina para outra através dos dispositivos móveis. Um vírus na sua definição bruta, não tem capacidade de propagar-se utilizando uma

rede de computadores, as redes estão nos domínios dos worms. Entretanto, o termo ”vírus” está sendo aplicado para malwares que deveriam ser considerados worms e o mesmo termo também está sendo utilizado para qualquer código que tenha a capacidade de auto-replicação.

Os vírus podem ser encontrados em vários estágios de replicação. Um *germ*, é a forma original de um vírus, ou seja, a forma anterior a qualquer replicação. Um vírus que falha ao replica-se é chamado de *intended*. A falha pode ser ocasionada por bugs do vírus, ou por exemplo, o vírus pode estar hosteado em um ambiente no qual não foi projetado para sua execução. Um vírus também pode encontrar-se no estado *dormant*, onde se encontra presente mas não consegue infectar o sistema, por exemplo, um vírus projetado para o sistema operacional Windows pode residir em um servidor de arquivo baseado no Linux e por isso permanece inativo, mas pode tornar-se uma ameaça quando transportado para máquinas com o sistema operacional Windows.

De acordo com (Erickson, 2008) um vírus se divide em 3 partes:

1. **Mecanismo de infecção**: Este mecanismo define como um vírus se espalha, como ele modifica os sistemas alvos. A técnica utilizada na infecção também é conhecida como vetor de infecção e um único vírus pode ainda ter vários vetores de infecção.

2. **Gatilho**: Decide quando liberará o payload.

3. **Payload**: Define o que o vírus faz, pode resultar em danos intencional ou acidental.

Das 3 partes que compõem um vírus, apenas o mecanismo de infecção não é opcional, pois é ele que define se o código é ou não um vírus. Por exemplo, na falta do mecanismo de infecção, o código seria uma bomba lógica. No pseudocódigo mostrado abaixo, a função de gatilho retorna um booleano que indica se o payload será carregado ou não.

```
def virus():
    infect()
    if trigger() is true:
        payload()
```

Lista de Listagens 4.3 Função de Gatilho de um Vírus

A função de infecção mostrada abaixo, seleciona um alvo e infecta-o, o alvo deve estar localmente acessível pelo vírus. O método utilizado para selecionar alvos pode variar

e a rotina de detecção destes alvos deve ser capaz de identificar se o alvo selecionado já foi previamente infectado, evitando desta forma uma reinfecção. A característica de poder identificar um alvo já infectado, auxilia os anti-vírus na busca destes códigos.

A rotina que realmente infecta o código, coloca alguma versão do vírus em algum lugar no sistema alvo.

```
Def infect():
    repeat k times:
      target = selectTarget()
      if no target:
         return
      infectCode( target )
```

Lista de Listagens 4.4 Rotina de Infecção de um Vírus

4.2.5 Worms

Um worm possui muitas características em comum com os vírus e é justamente por isso que muitas vezes um worm é classificado erroneamente como um vírus. A característica mais comum compartilhada por ambos é a capacidade de se auto-replicar, mas a replicação do worm possui duas diferenças principais em relação ao vírus. Primeiro, worms são auto-contido e não necessitam de nenhum outro código executável como seu hospedeiro. Segundo, worms espalham-se de máquina em máquina através da rede.

4.2.6 Rabbit

Quando um malware multiplica-se de maneira muito rápida, ele tende a ser identificado como um rabbit. O rabbit também é conhecido como *bactéria* pela mesma razão. Existem atualmente dois tipos de rabbit, o primeiro é um programa que consome grande quantidade de algum recurso da máquina, como por exemplo, memória e espaço em disco. Um ”fork bomb”, é um clássico exemplo deste tipo de rabbit, ele cria instâncias de novos processos em um loop infinito. O segundo tipo de rabbit é um tipo especial de worm, ele é um programa auto-contido que replica-se de máquina em máquina através da rede, mas com uma diferença, ele apaga a cópia original depois que uma replicação acontece, em outras palavras, só existe apenas uma cópia de um dado rabbit ativo na rede e esta cópia fica pulando de máquina em máquina.

4.2.7 Spyware

Este malware tem como característica principal a coleta de informações para serem transmitidas para alguém em algum lugar. As informações obtidas pelo spyware pode ser de vários tipos, mas na maioria das vezes incluirá informações que contenham grande valor, tais como:

- Nomes de usuários e senhas, coleta de arquivos na máquina infectada ou através de keyloggers.
- Endereços de email que são de grande valia para spammers.
- Conta de bancos e números de cartões de créditos.
- Chaves de licença de softwares para facilitar a pirataria.

Os vírus e os worms também tem a capacidade de coletar tais informações, mas o que diferenciam dos spywares e que estes últimos não tem a capacidade de auto-replicação.

O spyware pode infectar uma máquina de várias maneiras, pode vir embutido em um programa instalado pelo usuário ou explorando falhas técnicas nos web browsers. O ultimo método de infecção pode permitir que o spyware seja instalado simplesmente visitando uma web page e é desta forma chamado algumas vezes de *drive-by download.*

4.2.8 Adware

Adware possui características similares ao spyware e ambos são programados para obter informações sobre os usuários e também sobre seus hábitos. Tem a característica de ser muito focado em negócios e pode lançar *pop ups* avisando ou redirecionando o navegador do usuários para certas páginas, na esperança de realizar alguma venda. Alguns adwares tentarão direcionar os avisos no contexto das ações do usuário.

Adware pode também obter e transmitir informações sobre usuários para propósitos de negócios, como spywares, os adware não se auto-replicam.

4.2.9 Híbrido

Determinar o tipo exato de malware encontrado na prática não é uma questão trivial. A natureza do software possibilita criar malwares híbridos que possuem características pertencentes a vários tipos. Um exemplo clássico deste tipo de código foi apresentado por Ken Thompson (Aycock, 2006). Ele preparou um tipo diferente de compilador C, que além de compilar códigos em C, tinha as seguintes características adicionais:

1. Quando compilava códigos fontes que envolviam procedimentos de login, seu compilador inseria um back door para contornar o procedimento de autenticação.

2. Quando compilava o código fonte do compilador, ele produzia um executável com estas duas características

Na tentativa de enquadrar este código em alguma classificação de malware, poderíamos dizer que ele era um trojan horse, compilando e agindo de forma maliciosa em background, um vírus com capacidade de auto-replicação ou ainda um back door. Isto também demonstrou um mecanismo malicioso que pode ser perfeitamente inserido em um compilador.

4.2.10 Exploit

O código de um exploit é específico de uma vulnerabilidade em um conjunto de vulnerabilidades, seu objetivo é explorar alguma falha em software, procurando na maioria das vezes tomar o controle da máquina. Uma das maneiras mais comuns na inserção de exploits é através de programas que possuem a vulnerabilidade de estouro de pilha. O exploit pode ser utilizado tanto por profissionais da área de segurança quanto por atacantes mal intencionados, por isso, dependendo do uso do exploit, a exploração tende a ser maliciosa ou não.

4.2.11 Downloaders

Um downloader é um programa malicioso que instala um conjunto de outros itens na máquina sob ataque. Geralmente um downloader é enviado por e-mail e quando é executado, efetua downloaders de conteúdo malicioso oriundo de algum site, extrai e executa estes conteúdos.

4.2.12 Keyloggers

Um keylogger captura as teclas digitadas em um sistema comprometido, coletando informações sensíveis para um atacante. O keylogger é instalado no sistema sem o conhecimento do usuário e pode ficar imperceptível no sistema durante muito tempo.

4.2.13 Rootkits

Rootkits é um conjunto especial de ferramentas utilizadas por um atacante depois de uma invasão e um posterior acesso a nível de administrador. Geralmente o atacante invade um sistema utilizando exploits e instala versões modificadas de ferramentas comuns do sistema invadido. Tais rootkits são chamados de *user-mode rootkits*, porque as ferramentas modificadas executam em modo usuário.

Existem também rootkits sofisticados, como o **Adore**, que tem módulos componentes em nível de kernel, estas ferramentas são as mais perigosas porque modificam o comportamento do kernel. Esses rootkits conseguem esconder objetos tais como, processos, arquivos e chaves de registros das ferramentas que visam defender o sistema e também possuem a capacidade de camuflar outros componentes maliciosos.

5

Estratégias de infecção

5.1 Estratégias de infecção dos vírus

5.1.1 Infecção de boot

Um dos primeiros vírus de computador que obteve sucesso na sua infecção foi o vírus de boot. Em 1986 dois irmãos paquistaneses criaram o primeiro vírus de boot chamado *Brain*. Nos dias de hoje, a técnica de atacar o boot é raramente utilizada, mas é de grande valia entendermos como funcionava esta técnica, já que ela conseguia ter sucesso independente do sistema operacional utilizado na máquina alvo. O vírus de boot toma vantagem do processo de boot dos computadores, os computadores atuais possuem um tipo de memória que chamamos de ROM (Read Only Memory), que é uma memória muito diminuta é não comporta um sistema operacional para inicializar a máquina, por isso, existe a necessidade de carregar o sistema operacional de algum outro lugar, tal como, um disco ou uma rede.

Um típico disco rígido de computador é organizado em 4 partições, que dependendo do sistema operacional utilizado é identificada por letras, no caso da família Microsoft, ou por pontos de montagens, no caso da família Unix/Linux. O disco rígido é sempre dividido em cabeças, trilhas e setores. A MBR (Master Boot Record) esta localizada na cabeça 0, trilha 0 e setor 1, que representa o primeiro setor no disco rígido. Ela contém um código específico do processador para localizar nos registros da tabela de partição uma partição com seu setor de boot ativo. A tabela de partição esta armazenada na área de dados da MBR e logo a frente desta, se encontra um pequeno código chamado de *boot strap loader*.

A BIOS (Basic Input Output System) lê o primeiro setor de um disco, ou seja, a MBR, na sequência de boot especificada em sua configuração e quando tem sucesso,

executa o código do boot strap loader. O loader localiza as partições ativas na tabela de partição e carrega o primeiro setor lógico como setor de boot. O setor de boot contém código específico do sistema operacional. Este mecanismo possibilita aos computadores possuírem mais de uma partição com diferentes tipos de sistemas de arquivos e diferentes sistemas operacionais. Esta flexibilidade também auxilia no trabalho dos vírus, a MBR pode ser substituída por um vírus que só carrega a MBR original depois que seu próprio código esteja carregado na memória. A infecção da MBR é uma tarefa relativamente fácil para os vírus, o tamanho desta localização é de 512 bytes, somente pequenos códigos cabem neste local, mas este tamanho é o bastante para pequenos vírus.

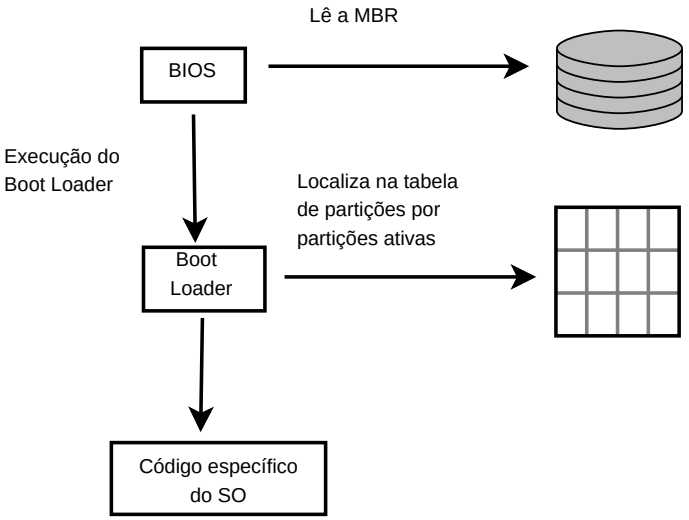

Figura 5.1 Sequência de Boot

5.1.2 Infecção de arquivos

Todos os sistemas operacionais possuem a noção do que seja um arquivo executável, um *file infector*, é um vírus que infecta estes tipos de arquivos, dentre os arquivos dito executáveis podemos ter: arquivos binários, arquivos batch e scripts. Na infecção de arquivos, duas questões apresentam grande relevância: *Onde o vírus é colocado? E Como o vírus é executado quando o arquivo infectado esta sendo executado?* Para responder a primeira questão temos as seguintes respostas:

Início do arquivo

Há muito tempo atrás, na época do MSDOS (MicroSoft Disk Operating System), existia um formado de arquivo binário muito simples com a extensão .COM, este formato tratava o arquivo inteiro como uma combinação de códigos e dados. Quando um arquivo .COM era executado, o arquivo inteiro iria para a memória e a execução começava saltando para o começo do arquivo carregado. Neste caso, para que o vírus tomasse o controle da execução, bastava colocar ele próprio no começo do arquivo. Vírus que agem desta forma são chamados de vírus *prepending*. A figura abaixo ilustra este técnica de infecção.

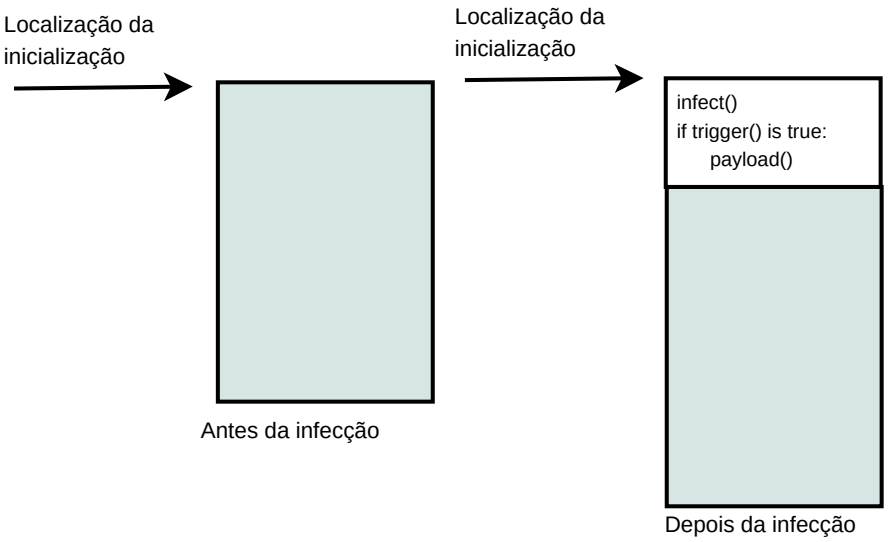

Figura 5.2 Vírus Prepending

Final do arquivo

De um outro lado, adicionar código no final de um arquivo é uma tarefa bem mais fácil, um vírus que coloca ele mesmo no final do arquivo é chamado de *appending vírus*. Existem duas possibilidades de se obter o controle da execução utilizando esta técnica. São elas:

- As instruções no código são armazenadas e substituídas por um salto para o código viral e posteriormente o vírus transferirá o controle de volta para o código infectado. O vírus pode tentar executar as instruções diretamente na localização armazenada ou restaurar o código de volta para sua localização original.

- A maioria dos formatos executáveis, por exemplo, o formato PE, especificam uma posição de início em seu cabeçalho. O vírus pode modificar este campo e apontar para seu próprio código que depois de executado pulará para o código original. A figura abaixo ilustra esta técnica.

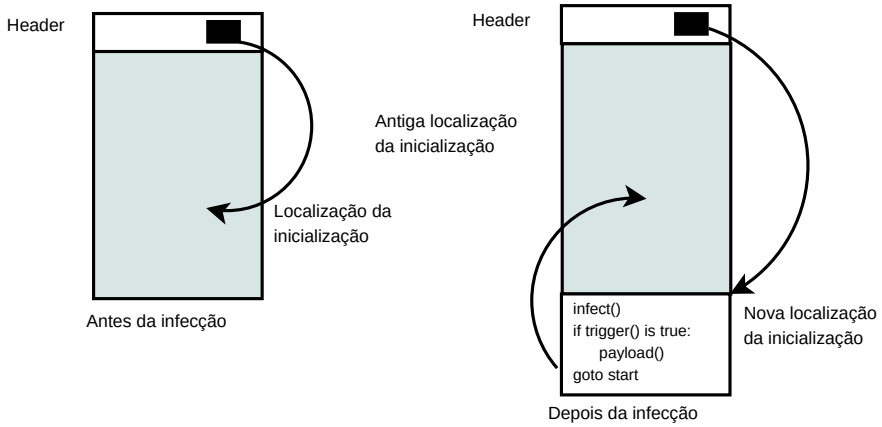

Figura 5.3 Vírus Apending

Sobrescrita do arquivo

Um vírus que sobrescreve um arquivo, coloca ele próprio em parte do código original, isto pode evitar o código adicional inserido pelas técnicas de appending e prepending, além disso, o vírus pode ser colocado no melhor lugar onde possa tomar o controle da execução. Para implementar esta técnica, existem várias opções com diferentes graus de complexidades e riscos. São elas:

- O vírus pode procurar e sobrescrever seções de valores repetidos no arquivo na esperança de evitar danos ao código original. Idealmente os valores substituídos devem ser restaurados logo após a execução do vírus

- Se o vírus for capaz de preservar o código substituído, então ele poderá alterar códigos de maneira arbitrária. O vírus pode utilizar arquivos externos, como arquivos de imagens, para preservar os dados alterados para depois restaurá-los.

- O vírus também pode utilizar os espaços super alocados nos arquivos executáveis. Partes de uma arquivo executável pode ser preenchido de maneira que ele fique alinhado nos limites das páginas de memórias, para que o sistema operacional

possa eficientemente mapeá-los. O resultado desta técnica de alinhamento deixa espaço nos executáveis onde o vírus pode se alocar.

- De outra maneira, um vírus pode comprimir o código original do arquivo na esperança de criar espaço para o seu próprio código e após a sua execução, o código original será descomprimido.

Fora do arquivo

Pertencente a esta categoria está o vírus denominado *companion*, ou companheiro. Este tipo de vírus não infecta nenhum arquivo na máquina alvo e toma vantagem do processo utilizado pelos sistemas operacionais na busca por um arquivo executável. Para ser executado e tomar o controle de uma máquina infectada, o vírus pode, por exemplo, alterar o ”path” padrão da procura de arquivos ou pode alterar alguma chave de registro do sistema operacional Windows.

5.1.3 Vírus de macro

Algumas aplicações tal como, processadores de textos, permitem que sejam incluídas dentro de seus textos macros, no intuito de automatizar algumas operações. Macros são pequenos pedaços de códigos escrito em uma linguagem na qual é tipicamente interpretada pela aplicação. Quando um documento que contém uma macro é carregado pela aplicação, as macros podem ser carregadas automaticamente, dando controle aos vírus de macros. Algumas aplicações avisam aos usuários sobre o perigo do carregamento de macros, mas na maioria das vezes estes avisos são ignorados.

Por utilizar uma linguagem muito simples e de certa forma mais portável do que muitas linguagens de programação, ficou muito mais fácil a criação e a pulverização de vírus utilizando as macros.

5.2 Estratégias de infecção dos worms

A estrutura geral de um worm pode ser descrita na seguinte forma:

```
Def worm():
    propagate()
    if trigger() is true:
       payload()
```

Lista de Listagens 5.1 Rotina de Infecção de um Worm

Pode-se perceber que em um nível de abstração, o worm é exatamente igual a um vírus e é por isso que em muitas vezes há uma confusão na hora de classificar um malware. A principal diferença está em como o código se propaga. Quando um código se propaga infectando um outro código, estaremos dentro do domínio de um vírus, mas se este código se propaga verificando vulnerabilidades em uma rede, estaremos no campo de domínio do worm.

Em muitos casos, a classificação de um worm acontece tomando como referência o seu modo de transporte, ou seja, se um worm utiliza o Instant Messaging para se propagar, então ele é chamado de *IM worm*, se utiliza o sistema de e-mail, então será classificado como *e-mail worm*.

O worm pode utilizar vários mecanismos para infectar uma vítima, dentre estes mecanismos estão, a engenharia social e a exploração de fraquezas técnicas. Para um usuário ser infectado por um worm que foi enviado por e-mail, ele não precisa executar o arquivo em anexo, existem vários worms que conseguem infectar simplesmente na abertura deste e-mail, utilizando para isso fraquezas como estouro de buffer. Um worm pode explorar transações legítimas existentes, é capaz de procurar e modificar comunicações de redes, pode esperar por transferências legítimas de arquivos executáveis, consegue substituir os arquivos transferidos por seu próprio código ou mesmo inserir-se no código como se fosse um vírus.

Os worms conseguem empregar várias técnicas que os vírus utilizam para tentar se camuflar no sistema infectado. Estes códigos podem ser oligomórfico, polimórfico ou metamórfico. Como citado anteriormente, a principal diferença entre um vírus e um worm é o seu modo de propagação, por isso, focaremos principalmente no mecanismo de propagação destes códigos. Segue abaixo uma tabela com os worms mais conhecidos e seus alvos de infecção.

Nome	**Descoberta**	**Infecção**	**Método de Execução**
WM/ShareFun	Fevereiro 1997	Documentos Word 6 e 7	Pelo Usuário
Win/RedTeam	Janeiro 1998	Arquivos NE do Windows	Pelo Usuário
W32/Ska@m	Janeiro 1999	WSOCK32.DLL	Pelo Usuário
W97/Melissa@mm	Março de 1999	Documento Word 97	Pelo Usuário
VBS/LoveLetter@mm	Maio 2000	Sobrescreve arquivos VBS	Pelo Usuário
W32/Nimda@mm	Setembro 2001	Arquivos PE 32 bits	Exploits

Tabela 5.1 Worms Mais Conhecidos e Seus Alvos de Infecção

5.2.1 Propagação

Uma característica muito importante de um worm é sua estratégia de se transferir para uma nova máquina e obter o controle do sistema. A maioria desses códigos assumem que o usuário tem um certo tipo de sistema, como o Windows, e envia um worm compatível com este sistema operacional.

O fator humano quase não tem nenhuma interferência na capacidade de propagação de um worm, por isso, ele tem a capacidade de se propagar em uma incrível velocidade. Em um extremo da escala de velocidade está os *fast burners*, que são os códigos que se propagam o mais rápido possível. Muitos desses worms possuem nomes especiais que refletem a sua velocidade de propagação. *Warhol worm*, infecta suas vítimas em menos de 15 minutos, *flash worm*, consegue realizar sua infecção em questão de segundos. O worm utiliza várias técnicas para conseguir esta velocidade de infecção, dentre elas citamos:

- Curto tempo de start.
- Minimiza a contenção entre instâncias de worm, evitando a contenção geral no tráfego da rede e a tentativa de reinfectar a mesma máquina.
- Utiliza escaneamento em paralelo para sondar possíveis vítimas.
- Utiliza protocolos com pouca sobrecarga de rede, tal como, o protocolo UDP (User Datagram Protocol), evitando na medida do possível protocolos pesados como TCP (Transmission Control Protocol), que exerce uma grande carga no estabelecimento de sua conexão.

No outro extremo da escala de velocidade estão os *surreptitious worms*, que para não serem notados, propagam-se lentamente. A propagação lenta pode ser utilizada para a construção de um ataque maciço de DDoS (Distributed Denial of Service) ou para a construção de uma botnet.

5.2.2 Semeamento inicial

Os worms necessitam ser injetados na rede e a maneira com que eles são inicialmente injetados é chamada de *seeding (semeamento)*. Um método efetivo de semeamento deve possuir as seguintes características: Deve ser o mais anônimo possível e deve ser capaz de distribuir muitas instâncias dos worms na rede. A seguir temos três possíveis métodos de semeamento.

1. Redes sem fio, atualmente ainda existem inúmeras redes sem fio sem nenhuma proteção, facilitando com isso o requisito de anonimidade requerido pelo semeamento, mas esta opção não é livre de risco, porque exige que o criador do worm esteja no campo de alcance da rede.

2. Spam, semear o worm através de spam satisfaz os critérios de efetividade, anonimidade e volume. O spam pode ser usado para semear um worm mesmo que este não se propague normalmente através de e-mail.

3. Botnets, semear o worm através de botnets também satisfaz o critério de efetividade. Botnets podem ser usadas para semear os worms através de spam e também pode liberar o malware diretamente em uma maneira altamente distribuída.

5.2.3 Encontrando alvos

As máquinas em uma rede são identificadas de duas maneiras: Pelo nome de domínio ou por seu endereço IP (Internet Protocol). Os nomes de domínios são fáceis de lembrar e são mapeados para IP de uma maneira transparente para o usuário. Endereços IP são apenas números usados para, por exemplo, no mecanismo de roteamento. Para a propagação de um worm, este tem que identificar possíveis alvos e é substancialmente mais fácil adivinhar um IP e encontrar um alvo, do que tentar adivinhar um nome de domínio exato.

O estratégia que um worm utiliza para encontrar possíveis máquinas alvos é chamado de scan, mas esta estratégia apesar do mesmo nome, é diferente da realizada pelos antivírus. Existem cinco estratégias básicas utilizadas pelos worms.

1. Escaneamento Randômico: O worm escolhe randômicamente os alvos, escolhendo um valor para ser usado como um endereço de IP.

2. Escaneamento com permutação: É onde as instâncias dos worms compartilham um permutação comum no espaço de endereçamento IP, o espaço é uma sequência pseudorrandômica dos possíveis valores dos endereços IP. Para cada nova instância, é dada uma posição na sequência onde será iniciada a infecção e o worm continuará a trabalhar. Se uma máquina encontrada já está infectada, então o worm escolhe uma nova posição na sequência randomicamente. Esta técnica possibilita ao worm um simples mecanismo de coordenação distribuída sem a sobrecarga de comunicação entre worms. Este mecanismo pode ser usado para detectar heuristicamente o nível de saturação das máquinas. Se um worm continuamente encontra

máquinas infectadas, isto pode indicar que a maioria das máquinas vulneráveis já foi infectada. O ponto de saturação pode também sinalizar que é o tempo oportuno de liberação do payload. Segue um exemplo para um escaneamento com permutação de 10 valores.

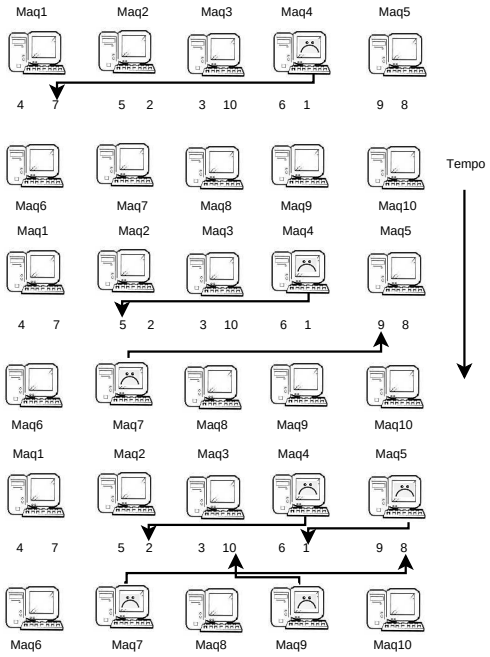

Figura 5.4 Escaneamento com Permutação

3. Escaneamento Localizado: Este tipo de escaneamento é muito bom para uma distribuição amplamente espalhada. Em um mesmo seguimento de rede, é muito provável que todas as máquinas sejam mantidas de maneira similar, ou seja, existe uma grande probabilidade de várias máquinas neste seguimento possuírem uma mesma vulnerabilidade. O escaneamento localizado, tenta tirar vantagem deste aspecto e de novo as máquinas são escolhidas randômicamente, mas agora a escolha é feita dentro de um determinado seguimento de rede.

4. Escaneamento com lista de alvos: Antes da liberação do worm, é confeccionada uma lista com IPs de algumas máquinas conhecidas que contém vulnerabilidades das quais o worm planeja explorar. Esta lista não precisa ser 100% exata, já que será utilizada apenas como ponto de partida. Depois que o worm é liberado, ele começa a atingir as máquinas na lista, a cada vez que a propagação do worm é realizada com sucesso, o restante da lista é dividida pela metade e esta metade é enviada junto com a nova instância do worm. Uma vez que a lista é exaurida,

o worm inicia outras estratégias de escaneamento. O escaneamento com listas de alvos evita a propagação por tentativa e erro e tem uma excelente propagação inicial. Uma variação deste esquema, pré-compila uma lista de todas as máquinas vulneráveis na internet e envia esta lista compressada junto com um worm.

5. Escaneamento topológico: Este tipo de escaneamento ocorre quando as informações em máquinas infectadas são utilizadas para selecionar novos alvos. A topologia pode ou não seguir a topologia física da rede, um worm pode, por exemplo, seguir uma informação obtida na interface de rede de uma máquina infectada ou mesmo na sua tabela de roteamento.

6. Escaneamento passivo: É um tipo de escaneamento que possibilita o worm escutar o tráfego de rede e obter informações sobre endereços válidos de IP, tipos e versões de sistemas operacionais, versões dos serviços de rede, padrões de tráfegos de redes e diante destas informações, ser capaz de escolher um adequado vetor de infecção.

5.2.4 Controle remoto e a interface de atualização

Uma outra importante característica de um worm é o seu canal de controle, é por este mecanismo que o escritor do worm pode enviar mensagens de controles para as máquinas infectadas, tais mensagens pode, por exemplo, instruir um ataque de DDoS em uma rede de zombie para atingir vários alvos desconhecidos.

A técnica mais empregada no controle remoto é baseada no uso de back door dentro do worm que se comunica diretamente a um host particular. Outras técnicas tais como, canais de IRC (Internet Relay Chat), também são utilizadas.

Algumas variantes de worms que ficam esperando por comandos dos atacantes, somente se propagam quando é enviado um comando de ”.spread” e quando isso acontece, os sistemas já comprometidos procuram por novos alvos para infectar.

5.2.5 Controle em rede P2P

Alguns worms implementam uma rede virtual com os nodos infectados para estabilizar operações de controle e comunicação. O worm Slapper é um exemplo desta categoria, ele usa o protocolo UDP e porta 2002 nas máquinas infectadas. Quando este worm infecta um novo alvo, ele passa para esta vítima o endereço de IP do atacante, cada novo

nodo recebe o endereço IP de todos os outros nodos e armazena em uma lista. Sempre que um novo IP é introduzido, todos os nodos atualizam as suas listas via conexão TCP.

A interface de controle do Slapper é muito avançada, o código foi emprestado do worm BSD/Scalper. O Slapper implementa um rede hierárquica que mantém a trilha de infecção do worm.

Uma interface de atualização também é uma importante característica e permite a atualização do código malicioso em um sistema já comprometido. Quando um atacante compromete um sistema com um particular exploit, evita-se infectar novamente o mesmo sistema, por isso, quando existe a necessidade de alterar o comportamento do worm o atacante utiliza a interface de atualização.

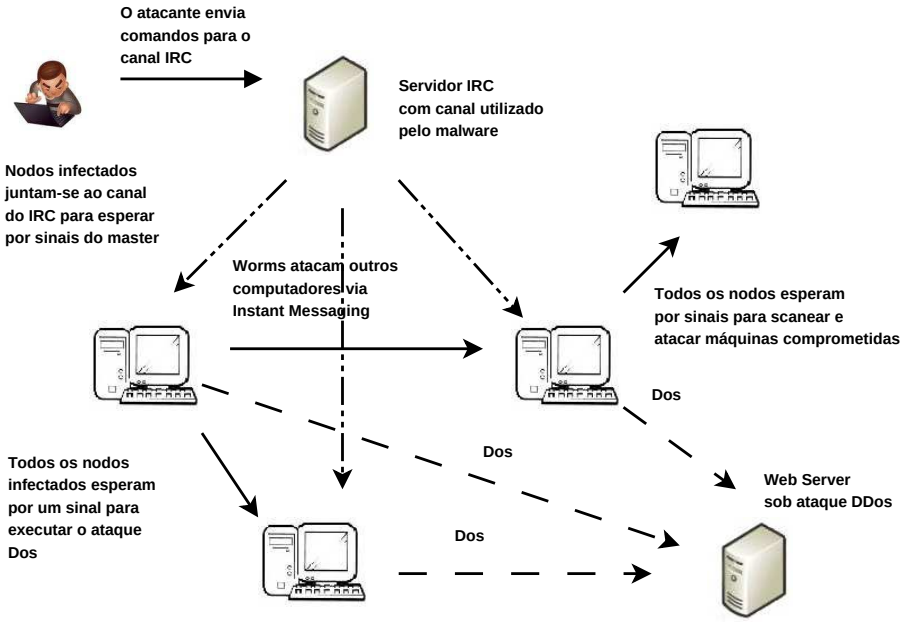

Figura 5.5 Controle Remoto do Worm Tendoolf

6

Estratégias de autoproteção

As estratégias de autoproteção são técnicas utilizadas pelos vírus para tentar prolongar a sua existência em um ambiente infectado. Estas técnicas tentam realizar as seguintes ações:

- Atacar agressivamente os softwares anti-vírus.
- Tentar dificultar as análises feitas pelos pesquisadores de anti-vírus.
- Tentar impossibilitar a sua detecção, conhecendo o funcionamento dos softwares anti-vírus.

Podemos sintetizar as tentativas de sobrevivência nas seguintes estratégias: *stealth, retroviruses, tunneling, ofuscação do ponto de entrada, ataques aos checadores de integridades, anti-emulação e blindagem de código.*

6.1 Stealth

Um vírus stealth age um pouco diferente dos outros vírus, ele tenta esconder a infecção manipulando informações solicitadas pelas aplicações. Para exemplificar como tais vírus trabalham, serão listadas abaixo duas técnicas utilizadas por eles.

- Depois de uma infecção o vírus restaura o timestamp do arquivo de modo que ele não pareça ter sido modificado.
- O vírus armazena de alguma maneira toda a informação pré-infecção, tais como, timestamp, tamanho do arquivo e seu conteúdo. Com essas informações, o vírus intercepta as operações de I/O através de interrupções ou bibliotecas compartilhadas e substitui o conteúdo modificado pelo original, quando é feita uma destas operações sobre o arquivo infectado.

Uma outra variação é um *reverse stealth* vírus que faz tudo parecer infectado, sendo o anti-vírus responsável pelo estrago no sistema.

6.2 Retroviruses

Um retrovírus é um vírus de computador que tenta de alguma forma dificultar ou impedir a operação normal de um anti-vírus ou de outros programas de segurança (Szor, 2005). Um retrovírus agressivo tem a capacidade de danificar um software anti-vírus, estando este no disco rígido ou na memória, dessa forma, a proteção pode ser desabilitada mesmo depois que uma máquina infectada é reinicializada. Uma maneira muito comum do funcionamento de um retrovírus é enumerar a atual lista de processos sendo executados na máquina infectada e matar qualquer processo que venha a ter, por exemplo, o nome de **Avgw.exe** ou **Zonealarm.exe**, outros tipos ações que podem ser executadas por este tipo de código malicioso são:

- Contornar ou eliminar qualquer firewall instalado na máquina infectada.
- Deletar o check de integridade das bases dos anti-vírus.
- Executar vírus via software anti-vírus.
- Prevenir os sistemas de conectar e fazer os downloads das atualizações dos anti-vírus via web sites.

6.3 Tunneling

O vírus tunneling tenta ser o primeiro na cadeia de chamadas das interrupções, eles se instalam na frente de todas as outras aplicações residentes e com isso conseguem chamar as interrupções diretamente no ponto de entrada de seus manipuladores originais. Obtendo primeiro as chamadas das interrupções, estes vírus conseguem obter o controle e depois repassar as chamadas para os manipuladores originais, contornando com isso, o monitoramento dos softwares anti-vírus. Um software anti-vírus pode frustrar esta estratégia se ele for capaz de se instalar no kernel do sistema operacional. Dentre as técnicas de tunneling podemos citar:

6.3.1 Rastreio com a interface de debugger

A ideia por trás desta técnica é anexa-se na INT 1, a interrupção do *single step* do debugger, mudar o flag de rastreio do processador para ON e executar uma chamada de interrupção inofensiva. Com este procedimento, a INT 1 será chamada toda vez que uma instrução for executada e o vírus pode rastrear o código até ele alcançar um particular manipulador.

6.3.2 Tunneling baseado em emulação de código

Este método usa a emulação de código para rastrear o caminho de execução até chegar no ponto de entrada de uma função desejada.

6.3.3 Uso de funções não documentadas

É o uso de funções não documentadas do sistema operacional para obter acesso aos manipuladores originais.

6.4 Ofuscação do ponto de entrada

Um vírus que tenta obter o controle de uma aplicação modificando o seu endereço de início ou o código que será executado no seu endereço de início, torna-se um alvo fácil para os anti-vírus que trabalham com heurísticas. Um vírus EPO (Entry Point Obfuscation), é um vírus que tenta obter o controle de uma aplicação atacando um lugar diferente de seu endereço de início, ele pode, por exemplo, ser executado na chamada da função API(Application Programming Interface) *ExitProcess*, que é uma chamada comum na saída de uma aplicação ou pode procurar por sequências de códigos repetitivas que são inseridas pelos compiladores, sobrescrever a sequência e pular para o código viral.

6.5 Ataques aos checadores de integridades

Os checadores de integridades são softwares que avisam quando um determinado arquivo sofreu algum tipo de modificação, estes softwares devem ser manipulados com muita prudência, por causa das altas taxas de falsos positivos emitidas por ele quando uma aplicação legítima faz uma alteração legítima. Os checadores de integridades podem possuir várias falhas que podem ser exploradas, uma das mais perigosas é quando

por algum motivo se força a recomputação do banco de dados dos checksums para todos oa arquivos

6.6 Anti-emulação

Os escritores de vírus perceberam que alguns scanners estavam utilizando emulação para detecção de seus códigos e começaram a criar ataques especificamente projetados para o emulador. Anti-emulação, são técnicas utilizadas pelo escritores de vírus na tentativa de frustrar a emulação de seus códigos maliciosos, basicamente dividem-se em 3: *Resistência, ser mais esperto e extrapolar o emulador.*

6.6.1 Técnicas de resistência

A quantidade de tempo que um emulador tem para emular um código é limitada quase exclusivamente pelo paciência dos usuários, então como um vírus poderia gastar o tempo do emulador a fim de conseguir fugir da detecção?

- Instruções que não fazem nada, tais como NOPs, poderiam ser adicionadas ao código do vírus e fazer com que o emulador gastasse seu tempo até sair, a partir deste momento, o código viral poderia ser executado.
- Um vírus poderia, por exemplo, agir benignamente 70% das vezes que executar, então o emulador só teria 30% de chance de classificar o código como malicioso.
- Técnicas de EPO e anti-heurísticas também se enquadram neste tipo de técnica.

6.6.2 Técnicas de ser mais esperto

Para que o código não pareça suspeito quando ele está sendo emulado, esta técnica espera até o exame minucioso do emulador acabar para restruturar o código viral. Espalhar o código de decriptação por toda parte, em vez de concentrar em um único local, é uma possibilidade de utilização desta técnica.

6.6.3 Técnicas de extrapolar o emulador

Um vírus pode de alguma forma tentar extrapolar os limites de um emulador na esperança que este apresente erros ou aborte sua execução. Para conseguir isso, um código poderia tentar:

- Utilizar-se de instruções não documentadas, na esperança do emulador não suportar as suas execuções.
- O sistema de memória do emulador poderia ser induzido a acessar uma localização de memória não usual na máquina real, causando uma condição de falta de memória.
- Recursos externos são quase impossíveis de serem emulados corretamente, um vírus pode aproveitar esta característica e procurar por estes recursos.

Abaixo seguem métodos práticos que utilizam as técnicas citadas acima:

Uso das instruções do coprocessador: Alguns escritores procurando por fraquezas nos emuladores, perceberam que a emulação do coprocessador não estava implementada e começaram a utilizar instruções do coprocessador.

Instruções MMX: Outros escritores foram mais longe, construindo vírus que faziam uso de instruções MMX e muitos emuladores não suportavam tais instruções.

Uso de manipuladores de exceção estruturada: Os vírus frequentemente configuravam um manipulador de exceção para criar uma armadilha para o emulador usado em produtos anti-vírus. Depois de configurar este manipulador, o vírus forçava a ocorrência de uma exceção. Durante a execução do código, o vírus indiretamente executava seu próprio manipulador, ganhando assim, o controle para seu decriptor polimórfico. Enquanto o emulador do anti-vírus não conseguir manipular a exceção, o código do vírus não poderá ser alcançado durante a emulação.

Execução randômica de código: Alguns vírus utilizam a execução randômica de seu ponto de entrada, em outras palavras, a carga do vírus não era garantida.

Uso de instruções de CPU (Central Processing Unit) não documentadas: Embora não existam muitas, mas ainda existe uma pequena parte de instruções de CPU não documentadas. Vários vírus utilizavam-se da instrução não documentada SALC no seu decriptor polimórfico como um lixo para interromper a emulação de certos produtos anti-vírus que não suportavam tal instrução.

Uso de força bruta na decriptação do código viral: A lógica é relativamente rápida quando se utiliza execução em tempo real, mas na emulação, gera-se longos loops, causando zilhões de interações, assegurando que o código real do corpo do vírus não será facilmente alcançado.

Uso de longos loops: Como no caso anterior, os vírus de computador usam longos loops na tentativa de derrotar decriptores genéricos que fazem uso da emulação. O loop é frequentemente um código polimórfico que não faz nada, mas as vezes este loop é utilizado para calcular uma chave de decriptação que é passada para o decriptor polimórfico.

6.7 Blindagem de código

6.7.1 Encriptação

Conforme (Stallings, 2010), criptografia é o estudo das técnicas que visam garantir o sigilo e/ou a autenticidade da informação. No que tange a encriptação de vírus, o objetivo é encriptar o corpo do vírus de uma maneira que dificulte a sua detecção, esta encriptação na maioria das vezes é relativamente fraca, exige pouco processamento para reverter o código e serve apenas para ofuscação. Até o vírus ser decriptado, ele não consegue ser executado, então neste tipo de código, existe a figura de um *loop decriptador* no qual decripta o vírus e transfere o controle para ele. Abaixo é apresentado o pseudocódigo de um vírus encriptado e a seguir os métodos que geralmente são feitas as encriptações.

```
key = 123
for i in 0..length( body ):
    body[i] = body[i] XOR key
    key = key + body[i]
```

Lista de Listagens 6.1 Rotina de Encriptação

1. **Simples encriptação**: Não se utiliza chaves e sim operações com bits, tais como, rotação, deslocamento, XOR, AND e etc.

2. **Encriptação com chaves estática**: Uma chave constante é utilizada para a encriptação e não é modificada a cada infecção.

3. **Encriptação com cifras de substituição**: Utiliza uma tabela de mapeamento entre as formas encriptada e decriptada.

4. **Encriptação forte**: Algumas vezes os vírus utilizam uma encriptação forte para fugir da detecção e esta operação hoje em dia, tende a ser facilitada por causa das inúmeras bibliotecas contendo algoritmos criptográficos.

Todas as técnicas listadas acima possui a mesma deficiência na hora do vírus esconder-se da detecção, a deficiência cai na assinatura criada pelo corpo encriptado. Para evitar esta deficiência, o vírus terá que, a cada infecção, criar randomicamente as chaves criptográficas, encriptar o corpo do vírus e atualizar o loop decriptor com a nova chave criptográfica.

6.7.2 Oligomorfismo

Se a cada infecção o vírus utiliza uma nova chave criptográfica, então o corpo do vírus encriptado também será diferente, mas o problema é que a rotina utilizada para decriptar este código permanece a mesma, deixando um padrão de detecção para os anti-vírus.

Um vírus oligomórfico é um vírus encriptado que possui um pequeno e finito número de diferentes rotinas para decriptar o código malicioso. Em termos de detecção, um vírus oligomórfico somente torna a detecção um pouco mais difícil, o software anti-vírus deve possuir todas as possíveis rotinas de decriptação e testar todas elas.

6.7.3 Polimorfismo

Um vírus polimórfico é bem parecido com um vírus oligomórfico, ambos são encriptados e ambos modificam seu loop de decriptação a cada infecção, a principal diferença entre eles é que para fins práticos um vírus polimórfico possui uma infinidade de possíveis loops de decriptação. Diante destas características, surgem dois desafios a serem vencidos, como um vírus deste tipo pode detectar se uma arquivo já foi infectado por ele anteriormente e como é o processo de modificação do seu loop de decriptação a cada infecção. Para responder a primeira pergunta, os escritores de vírus utilizam as seguintes técnicas:

- **Timestamp do arquivo**: Utiliza o timestamp dos arquivos de modo que a soma dos atributos de hora e data seja sempre uma constante K para todas as infecções.
- **Tamanho do arquivo**: Torna o tamanho de um arquivo múltiplo de alguma constante K.
- **Lacunas em arquivos executáveis**: Nem todas as partes de um formato de arquivo executável são utilizadas pelo sistema operacional, então o vírus pode utilizar uma área que geralmente não é usada e colocar algum tipo de flag.
- **Atributos no sistema de arquivos**: Pode ser utilizado algum atributo do sistema de arquivo para indicar a infecção.

- **Armazenamento externo**: Pode ser utilizado algum meio de armazenamento externo, tal como, chaves do registro do Windows, para indicar infecções.

É interessante notar que estas técnicas não precisam ser 100% eficazes, o máximo que pode ocorrer no caso de um falso positivo é a não infecção de um arquivo.

Com relação a segunda pergunta, o vírus polimórfico utiliza um engenho de mutação, este engenho toma como entrada uma sequência de códigos e resulta como saída uma outra sequência de códigos diferente, mas equivalente a sequência de entrada. Segue abaixo algumas transformações utilizadas pelo engenho de mutação:

- **Equivalência de instrução**: Frequentemente arquiteturas como a CISC possuem diferentes instruções que causam o mesmo efeito.
- **Equivalência de sequência de instruções**: Instruções diferentes que quando agrupadas possuem o mesmo efeito.
- **Reordenamento de instruções**: Reordena cadeias de instruções que não possuem dependências entre suas instruções individuais.
- **Troca de registradores**: Operação simples, mas que modifica o padrão de bits que codifica as instruções.
- **Reordenação de dados**: Modificar as localizações dos dados tem o mesmo efeito das trocas de registradores.
- **Inserção de código ”junk”**: O termo ”junk”, indica instruções que são inseridas no código final, mas que não afetam o resultado do código original.

6.7.4 Metamorfismo

Diferentes de muitas categorias de vírus já citadas, os vírus metamórficos não são encriptados e por isso não apresentam um loop decriptor. Estes vírus criam uma nova versão de seu corpo a cada nova infecção e assim conseguem enganar os anti-vírus. Operando em um código intermediário, o engenho de mutação dos vírus metamórficos desfaz velhas ofuscações, aplica novas transformações e gera um novo código de máquina.

Enquanto os vírus polimórficos e metamórficos são muito difíceis de detectar, eles também são muito difíceis de escrever, por este motivo, o numero de vírus pertencentes a estas categorias são pequenos em comparação com outras categorias.

6.7.5 Códigos auto-modificáveis

Uma possibilidade para um atacante colocar dificuldades durante a análise, é utilizar algum conjunto de códigos auto-modificáveis. Quando o código é examinado em um disassembler, este torna-se de difícil leitura. Abaixo segue um pequeno exemplo de como poderia ser empregada esta técnica.

```
Código original:
MOV CX, 100h ; CX = 100h
MOV AH, 40h  ; AH = 40h
INT 21h      ; Aplica uma Interrupção

Código modificado:
MOV  CX, 003Fh ; CX = 003Fh
INC  CX        ; CX = CX + 1, CX = 0040h
XCHG CH,CL     ; Troca CH e CL, CX = 4000h
XCHG AX,CX     ; Troca AX e CX, AX = 4000h, AH = 40h
MOV CX, 0100h  ; CX = 100h
INT 21h        ; Aplica uma Interrupção
```

Lista de Listagens 6.2 Código Auto-Modificável

6.7.6 Uso de checksum

Vários vírus utilizam algum tipo de checksum, como por exemplo, o algoritmo CRC32, para evitar usar strings que possam fornecer dicas sobre código. Alguns checksums tornam o significado do código extremamente difícil.

6.7.7 Compressão de código

Os atacantes frequentemente utilizam compressão de código porque ela possui a vantagem de possibilitar a utilização de códigos maiores. A análise do código torna-se mais difícil porque o executável precisa ser descomprimido primeiro. Além disso, esta técnica também dificulta o trabalho dos scanners e dos analisadores baseados em heurísticas.

6.7.8 Anti-heurísticas

Análise heurística pode detectar vírus desconhecidos e também variantes de vírus utilizando métodos estáticos e dinâmicos. *A heurística estática* confia no formato do arquivo

e também nos seguimentos existentes nestes formatos, enquanto a *heurística dinâmica*, utiliza emulação de código para imitar o processador e o ambiente do sistema operacional com o objetivo de detectar operações suspeitas. A primeira geração de heurísticas obteve um grande sucesso contra os vírus que infectavam arquivos PE do Windows. Mas não demorou muito para os escritores de vírus perceberem este sucesso e implementar ataques contra os detectores de heurísticas. Vários tipos de infecção anti-heurísticas foram implementadas durante anos, abaixo segue exemplos de como os escritores de vírus utilizavam estas técnicas.

Infecção das áreas de sobras na primeira seção: Alguns vírus não modificam o ponto de entrada de uma aplicação para apontar para a ultima seção, em vez disso, o vírus sobrescreve as áreas de sobra da primeira seção e colocam instruções que saltam para o início do código do vírus.

Infecção da primeira seção através do deslocando das outras seções: Alguns vírus são grandes demais e seus códigos não caberiam nas áreas de sobras, então o vírus cria um espaço para ele próprio deslocando cada seção do PE para depois de seu código. O vírus adiciona seu próprio código e ajusta o *raw data offset* de cada seção subsequente por um tamanho de 512 bytes.

Infecção da primeira seção através de compressão: Com esta técnica, o vírus comprime a seção de código da aplicação e coloca o seu código junto com ela. No momento da execução da aplicação, o vírus executa a descompressão do código comprimido.

Seleção de um ponto de entrada randômico na seção de código: Algumas vezes, os vírus procuram a seção de relocação das aplicações, no intuito de encontrar uma posição segura para executar algumas substituições de códigos na seção de código de um arquivo PE. Os vírus modificam certas instruções CALL e PUSH para instruções JMP que pulam para o ponto de início do código do vírus.

Substituição das áreas de alinhamento dos compiladores: Alguns vírus substituem com seus códigos as áreas de alinhamentos dos compiladores que são preenchidas com zeros ou 0xCCs.

Não deixar as seções com o atributo de escrita: Seções com o atributo de escrita ligado são alvos fáceis para heurísticas estáticas, então, se o vírus precisar alterar um seção que não tenha o atributo de escrita ligado ele deve ligar, alterar o código e não se esquecer de desligar este atributo.

Renomear as seções existentes: Alguns scanners de primeira geração tentam checar se uma seção conhecida obteve o controle durante uma emulação. Alguns vírus modificam o nome das seções para uma string randômica, como resultado, a heurística estática não descobre o vírus facilmente baseado no nome das seções.

6.7.9 Anti-disassembly

Os vírus de computadores escritos em linguagem assembly são difíceis de entender, porque eles frequentemente utilizam truques que programas escritos em outras linguagens raramente usam. Os maiores ataques nos disassemblers são as simples variações das técnicas de ofuscação, tais como, encriptação, polimorfismo e especialmente o metamorfismo.

Conforme (Singh, 2009), anti-disassembly é um método anti-reversing usado para escapar dos disassemblers. Isto faz com que os disassemblers gerem códigos incorretamente revertidos.

Anti-disassembly utiliza códigos ou dados especialmente trabalhados em um programa para fazer com que as ferramentas de disassembly produzam uma listagem incorreta do código analisado. Esta estratégia é trabalhada por escritores de vírus manualmente como uma técnica separada (Sikorski and Horniq, 2012). Autores de malwares utilizam a técnica de anti-disassembly para retardar ou prevenir a análise de seus códigos e incrementam o nível de conhecimentos necessários para um analista de malware. Além de retardar e prevenir a análise humana, esta técnica também é muito efetiva em frustrar técnicas de análises automáticas.

Entendendo a técnica

As técnicas de anti-disassembly foram criadas a partir das fraquezas dos *algoritmos de disassembly*. Quando um escritor de malware implementa uma técnica anti-disassembly, ele cria uma sequência de código que faz com que o disassembler mostre uma lista de instruções que diferem das instruções que serão realmente executadas. Qualquer disassembler deve fazer certas suposições para tentar criar o código revertido, e quando estas suposições falham, o escritor de malware tem a oportunidade de enganar o analista de malware.

Existem dois tipo de algoritmos de disassembly: *O linear e o orientado a fluxo*.

O algoritmo linear

A estratégia do algoritmo linear é iterar sobre o bloco de código, disassemblando uma instrução após outra, de maneira linear, sem considerar qualquer operação de desvio. Esta é uma estratégica básica que é usada amplamente em debuggers. O algoritmo linear usa o tamanho das instruções revertidas para determinar qual byte será o próximo a ser revertido, sem se importar com as instruções de fluxo de código.

O método que os autores de malware exploram os algoritmos linear de disassembly, confia em colocar em seu código bytes de dados que formam *opcodes* de instruções multibytes. Estes algoritmos são fáceis de serem vencidos, porque eles são incapazes de distinguir o que é código e o que é dado. Abaixo segue o fluxo lógico deste algoritmo:

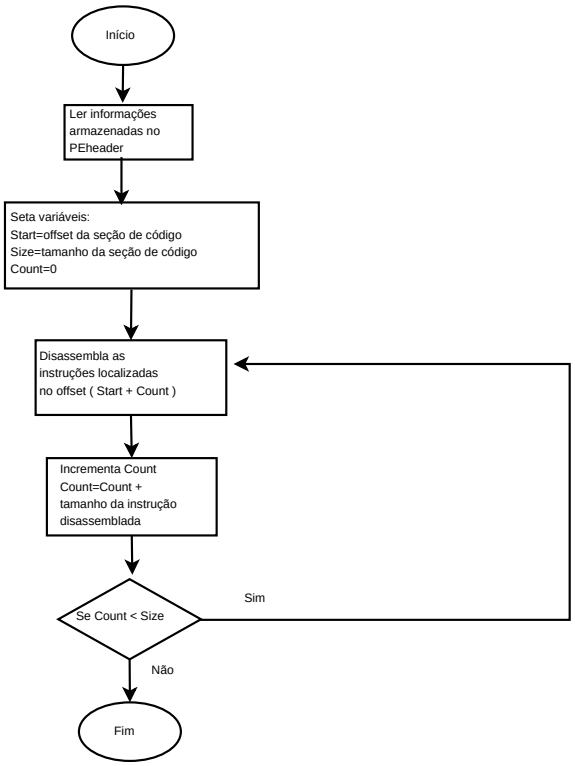

Figura 6.1 Fluxo Lógico do Algoritmo Linear de Disassembly

O algoritmo orientado a fluxo

Estes algoritmos fazem parte de uma categoria mais avançada de algoritmos de disassembly. Este método é utilizado nos principais disassemblers, tal como o IDA Pro.

A principal diferença entre os algoritmos linear e o orientado a fluxo, é que este

último não itera cegamente sobre o código, ele examina cada instrução, considerando o fluxo do código e constrói uma lista de localizações a serem revertidas. Abaixo segue o fluxo lógico deste algoritmo:

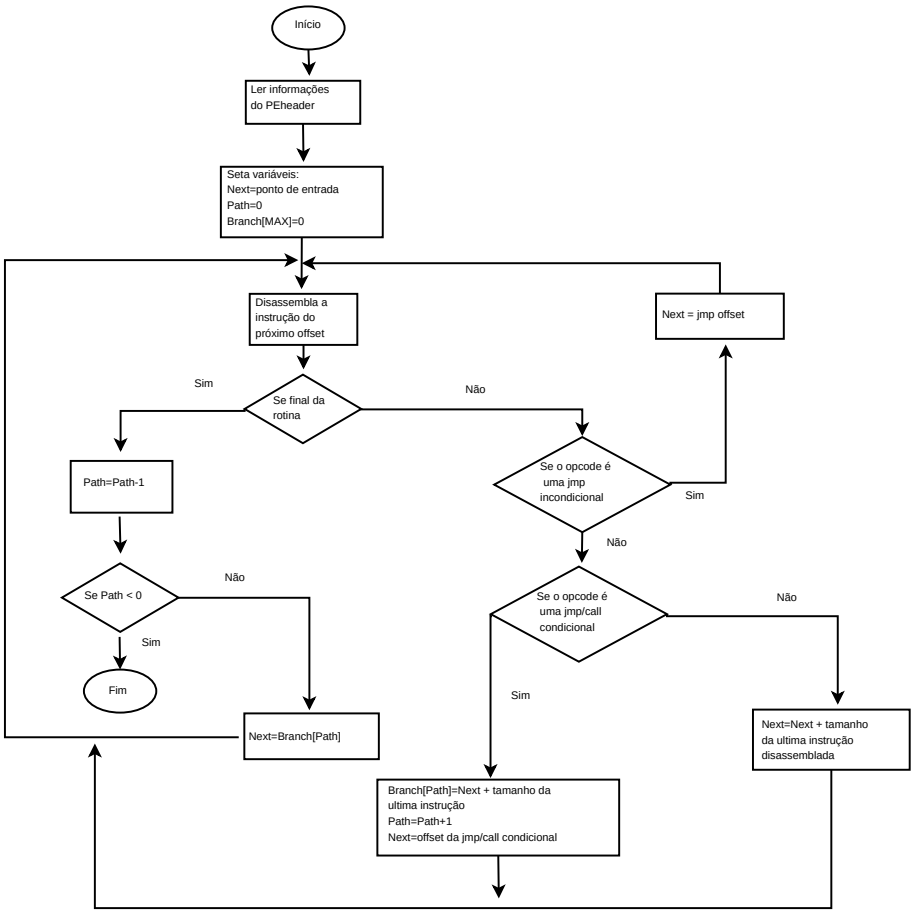

Figura 6.2 Fluxo Lógico do Algoritmo Orientado a Fluxo de Disassembly

Linear x Orientado a fluxo

Para demonstrar como estes algoritmos trabalham na prática, e como seria o resultado final de um disassembly utilizando os dois algoritmos, será utilizado um exemplo do (Sikorski and Horniq, 2012), que pode fornecer uma ideia das vantagens e desvantagens encontradas nestes algoritmos.

O seguinte trecho de código somente pode ser corretamente revertido ser for utilizado um algoritmo orientado a fluxo.

```
TEST EAX, EAX
```

```
        1:JZ short loc_1A
        2:PUSH Failed_string
        3:CALL printf
        4:JMP short loc_1D
;---------------------------------
Failed_string: db ''Failed'',0
;---------------------------------

5:loc_1A:
            XOR EAX, EAX
loc_1D:
            RETN
```

Lista de Listagens 6.3 Código de Teste Algoritmo Linear x Orientado a Fluxo

Este exemplo começa com uma instrução TEST e uma instrução JZ condicional. Quando o disassembler orientado a fluxo alcança a instrução condicional JZ na linha 1, ele nota que em algum ponto no futuro será necessário reverter a localização ”loc_1A” na linha 5 e adicionará este alvo na lista de lugares para reverter no futuro. Porque existe somente um ramo condicional, a instrução na linha 2 é uma possibilidade na execução, então o disassembler também reverterá ela também.

As linhas 2 e 3 são responsáveis por imprimir a string ”Failed” na tela. Seguindo o código, o disassembler orientado a fluxo adicionará o alvo ”loc_1D” na lista de lugares para reverter no futuro. Já que a instrução JMP é incondicional, o disassembler não reverterá automaticamente a instrução imediatamente seguinte na memória. Em vez disto, ele voltará e checará a lista de lugares que ele anotou previamente, tal como ”loc_1A”, e reverterá a partir deste ponto.

Em contraste, quando um disassembler linear encontrar a instrução JMP, ele continuará cegamente revertendo as instruções sequencialmente na memória, independentemente do fluxo lógico do código. Neste caso, a string ”Failed” será revertida como código, inadvertidamente escondendo a string ASCII (American Standard Code for Information Interchange) e as ultimas duas instruções no fragmento de código. Segue abaixo o resultado de um disassembler linear tomando como entrada o fragmento de código acima.

```
TEST EAX, EAX
JZ short near ptr loc_15+5
PUSH Failed_string
CALL printf
```

```
                JMP short loc_15+9
Failed_string:
                INC ESI
                POPA
loc_15:
                IMUL EBP, [EBP+64h], 0C3C03100h
```

Lista de Listagens 6.4 Código Resultado do Algoritmo Linear

Na reversão linear, o disassembler não faz escolhas sobre quais instruções ele irá reverter, enquanto na reversão orientada a fluxo, o disassembler faz escolhas e suposições.

Técnicas anti-disassembly

A estratégia principal utilizada pelos malwares para forçar o disassembler a produzir uma reversão errada, é tirar vantagens das escolhas e suposições feitas pelo disassembler. A maioria das técnicas de anti-disassembly podem ser consertadas manualmente por um analista experiente.

Instruções de saltos com condições constantes

Uma técnica comum encontrada é composta de uma simples condição de salto colocada em um lugar onde a condição sempre será satisfeita. Se tivermos o seguinte código:

```
XOR EAX, EAX
JZ loc
...
```

Lista de Listagens 6.5 Código de Salto com Condição Constante

No código acima, a instrução XOR, conforme (Irvine, 2007) e (Manzano, 2007), colocará o valor zero no registrador EAX e setará o flag *zero*, como a instrução JZ salta apenas se o flag *zero* estiver setado, isto faz com que a instrução fique incondicional, o disassembler não percebe o truque e continua a reverter um código que não será executado.

Instruções de saltos para o mesmo alvo

Uma outra técnica comumente encontrada é colocar duas instruções de saltos adjacentes pulando para o mesmo alvo. Se for colocado um instrução *JZ loc* seguida de *JNZ loc*, a localização ”loc” sempre será alcançada. A combinação das JZ e JNZ adjacentes

com a mesma localização tem um efeito de uma instrução JMP incondicional, mas o disassembler não percebe porque ele só reverte uma instrução por vez. Quando o disassembler encontra a instrução JNZ, ele continua revertendo, independente do fato de que as instruções seguintes ao JNZ nunca serão executadas na prática.

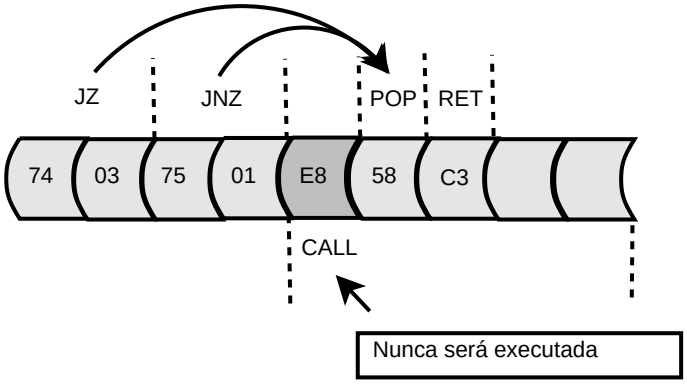

Figura 6.3 Instruções de Saltos para o Mesmo Alvo

Impossível disassembly

Isto ocorre quando encontramos um cenário onde um dado byte pode ser parte de múltiplas instruções que são executadas. Nenhum disassembler existente no mercado representará um simples byte como parte de duas instruções.

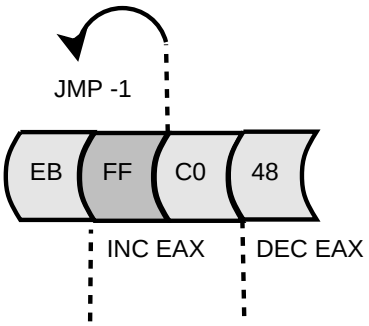

Figura 6.4 Salto para o Interior da Instrução

O dilema quando se tenta reverter a instrução acima é que, se o disassembler escolher representar o byte FF como parte da instrução JMP, então ela não será mostrada como o início da instrução INC EAX. O byte FF é parte de ambas instruções e os modernos disassemblers não tem como representar isso. Este tipo de técnica pode ser inserida em qualquer posição do código e quebrará a cadeia de validade da reversão.

6.7.10 Anti-debugging

Os escritores de malwares tem ciência de que os analistas de malwares utilizam as ferramentas de debugging para descobrir como os códigos funcionam. As técnicas anti-debugging são utilizadas para retardar e dificultar o processo de análise de códigos. Uma vez que o malware percebe que está sendo executado dentro de um debugger, ele altera a execução normal do código ou simplesmente aborta a execução.

Detecção de debugger no Windows

Várias técnicas são utilizadas pelos malwares para detectar a presença de um debugger, estas técnicas incluem o uso da API do Windows, o check manual de estruturas e a procura por resíduos nos sistemas deixados pelo debugger.

Uso da API do Windows

Existem várias funções na API do Windows que são capazes de detectar a presença de um debugger. Algumas são específicas para a detecção destas ferramentas e outras, apesar de serem projetadas para diferentes finalidades, também conseguem fornecer informações sobre a presença de um debugger. As seguintes funções da API do Windows podem ser empregadas para técnicas anti-debugging:

IsDebuggerPresent: Esta função acessa a estrutura do PEB (Process Environment Block) e pesquisa pelo campo *IsDebugged*, no qual retornará um valor zero, se o programa não estiver sendo executado em um contexto de debugger ou retornará um valor diferente de zero na presença de um debugger.

CheckRemoteDebuggerPresent: Esta função também acessa o PEB para pesquisar o campo *IsDebugged*, porém, ela pode tanto consultar o seu ambiente de processo como um outro na mesma máquina local. Esta função recebe um manipulador de processo como parâmetro e checa se ele tem um debugger anexado.

NtQueryInformationProcess: Esta função recebe dois parâmetros: um manipulador de processos e um outro que indica que tipo de informação será consultada, se esta função for executada com o segundo parâmetro igual a *ProcessDebugPort*, ela terá condições de informar se um debugger está anexado ao processo consultado.

OutputDebugString: Esta função é utilizada para enviar uma string que será mostrada no ambiente do debugger. Se esta função é chamada dentro de um contexto de

debugger, ela será executada com sucesso, caso contrário, ela falhará. Com este tipo de teste, o malware detectará a presença de um debugger, mesmo com uma função que não é específica para isso.

Check manual de estruturas

A estrutura PEB do Windows é mantida pelo sistema operacional para cada processo em execução, ela contém todos os parâmetros à nível de usuário associados com cada processo. Estes parâmetros incluem dados, variáveis de ambientes, módulos carregados, endereços na memória e informações sobre debuggers. Segue abaixo a estrutura PEB do Windows:

```
typedef struct _PEB {
  BYTE Reserved1[2];
  BYTE BeingDebugged;
  BYTE Reserved2[1];
  PVOID Reserved3[2];
  PPEB_LDR_DATA Ldr;
  PRTL_USER_PROCESS_PARAMETERS ProcParameters;
  BYTE Reserved4[104];
  PVOID Reserved5[52];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcInitRoutine;
  BYTE Reserved6[128];
  PVOID Reserved7[1];
  ULONG SessionId;
} PEB, *PPEB;
```

Lista de Listagens 6.6 Estrutura PEB do Windows

O método de se utilizar a API do Windows para a detecção de debuggers é muito direto, porém, isto é muito fácil de ser reconhecido por um processo de heurística estática, por isso, os autores de malwares preferem acessar manualmente as estruturas em busca de informações sobre debuggers. Em um check manual, vários campos dentro da estrutura do PEB podem informar sobre a presença de um debugger. Seguem abaixo vários campos que podem ser consultados para confirmar a presença de um debugger:

BeingDebugged : Quando um processo está rodando, a localização do PEB pode ser referenciada pela localização *fs:[30h]*. As duas listagens de códigos a seguir podem ser utilizadas para consultar o campo *BeingDebugged* da estrutura PEB.

```
mov eax, dword ptr fs:[30h]
mov bl, byte ptr [eax+2]
test bl, bl
jz NoDebuggerDetected
```

Lista de Listagens 6.7 Método mov para Consulta do BeingDebugged

```
push dword ptr fs:[30h]
pop edx
cmp byte ptr [edx+2], 1
je DebuggerDetected
```

Lista de Listagens 6.8 Método push/pop para Consulta do BeingDebugged

A ideia é muita simples: primeiro coloca-se a localização da estrutura PEB em um registrador, soma-se ao offset o valor 2 e testa se a posição contém um valor diferente de zero; se este for o caso, a presença de um debugger é confirmada.

ProcessHeap : Este campo é indicado por uma localização não documentada dentro da array *Reserved4*, esta localização contém o primeiro heap do processo alocado pelo loader do Windows. O ProcessHeap está localizado na posição 0x18 da estrutura PEB e contém campos utilizados para informar ao kernel se o heap foi criado dentro de um debugger. Estes campos são conhecidos como: *ForceFlags* e *Flags*. Esta técnica pode ser utilizada da seguinte forma:

```
mov eax, large fs:30h
mov eax, dword ptr [eax+18h]
cmp dword ptr ds:[eax+10h], 0
jne DebuggerDetected
```

Lista de Listagens 6.9 Método para Consulta do ProcessHeap

O código acima move o PEB para o registrador EAX, acessa o ProcessHeap através do offset EAX+18h e por último consulta o *ForceFlags* através do offset EAX+10h.

NTGlobalFlag : A informação que o sistema utiliza para determinar a criação das estruturas de heap está armazenada em uma localização não documentada do PEB, no offset 0x68. Se o valor desta localização for igual a 0x70, então a presença do debugger é confirmada. Esta técnica pode ser utilizada da seguinte forma:

```
mov eax, large fs:30h
cmp dword ptr ds:[eax+68h], 70h
jz DebuggerDetected
```

Lista de Listagens 6.10 Método para Consulta do NTGlobalFlag

O código acima move o PEB para o registrador EAX e compara o NTGlobalFlag na posição EAX+68h com o valor 0x70.

Checando resíduos no sistema

Os debuggers costumam deixar resíduos no sistema operacional. Os resíduos, tal como certas chaves de registros, podem ser utilizados pelos malwares para determinar se um debugger está sendo utilizado. A chave *HKEY_LOCAL_MACHINE\SOFTWARE \Microsoft\Windows NT\CurrentVersion\AeDebug* pode ser utilizada para denunciar a presença de um debugger. Além das chaves, os malwares também podem procurar por arquivos e diretórios no sistema que também denunciam a existência de debuggers.

Identificando o comportamento do debugger

Segundo (Justin Ferguson and Pearce, 2008), os debuggers podem ser usados para setar breakpoints no código ou caminhar passo a passo através de um processo em execução, no intuito de ajudar o analista no seu trabalho de análise. Quando um debugger seta um breakpoint ou caminha sobre o código, ele faz várias modificações no código analisado. Várias técnicas anti-debugging, tais como, escaneamento de interrupções, ckecksums de código e checks de timing podem ser utilizadas para detectar o comportamento dos debuggers.

Escaneamento de interrupções

Quando um breakpoint é setado, o debugger utiliza a interrupção de software *Int 3* para temporariamente substituir uma instrução no programa em execução e chamar o manipulador de exceção do debugger. O opcode da *Int 3* que substitui as instruções é o **0xCC**, e os malwares que utilizam esta técnica anti-debugging fazem buscas em seu código para encontrar este opcode.

Checksums de código

Quando é setado um breakpoint ou quando é feita uma navegação passo a passo pelo código, o debbuger modifica o código do malware, os malwares para detectar a presença

de um debugger, simplesmente executam um CRC (Cyclic Redundancy Check) ou um MD5 (Message-Digest algorithm 5) de seu próprio código a fim de detectar estas modificações.

Check de timing

O check de timing é uma das maneiras mais utilizadas pelos malwares para detectar debuggers, isto acontece porque o processo executa mais lentamente quando analisado dentro de um debugger. Os seguintes métodos de checks de timing são utilizados para detectar um debugger:

- Registrar um timestamp, executar algumas operações, registrar em seguida um outro timestamp e por fim comparar os dois timestamps, se o resultado for muito grande é provável a existência de um debugger.
- Registrar um timestamp antes e depois do lançamento de uma exceção. Se o processo não estiver sendo debugado, a exceção será manipulada rapidamente.

A instrução RDTSC

Uma outra técnica de check de timing faz uso da instrução *rdtsc*, que retorna um contador de ticks desde o último reboot da máquina. Mas uma vez o malware simplesmente executa esta instrução duas vezes e compara a diferença das duas leituras, se o valor for muito alto, é provável que haja um debugger.

Uso da função API QueryPerformanceCounter

Esta função confia no fato de que existem registradores que armazenam contadores das atividades do Processador. Como a instrução rdtsc, o malware simplesmente executa esta instrução duas vezes e compara a diferença das duas leituras, se o valor for muito alto, é provável que haja um debugger.

Influenciando no funcionamento do debugger

Malwares as vezes utilizam técnicas que interferem no funcionamento normal do debugger. Técnicas como, chamadas TLS (Thread Local Storage) e inserção de interrupções, podem interromper a execução de um malware que estiver sendo analisado por um debugger.

Chamadas TLS: A seção do ponto de entrada é o nome de uma seção nos arquivos PE que contém o campo AddressOfEntryPoint (Michael Hale Ligh and Richard, 2011). A maioria dos debuggers iniciam a aplicação tomando com referência o endereço colocado neste campo. Uma chamada TLS pode ser usada para executar um código antes do ponto de entrada e assim executar secretamente em um debugger. TLS é uma classe de armazenamento e permite que threads diferentes possam manter valores diferentes para uma variável declarada como TLS. Quando a TLS é implementada por um executável, seu código conterá uma seção *.tls* no cabeçalho do arquivo PE.

Inserção de INT 3: A INT 3 é utilizada pelos debuggers para inserir breakpoints e isto consiste em inserir no código analisado o opcode 0xCC. Uma técnica utilizada pelos malwares é inserir o opcode 0xCC dentro de seções válidas do código a fim de fazer com que os debuggers pensem que são breakpoints.

Inserção de INT 2D: Esta técnica funciona da mesma forma que a INT 3, mas faz uso de um opcode que é utilizado em debuggers de nível kernel.

Inserção da ICE: ICE é uma instrução não documentada da Intel. Ela gera um exceção de navegação passo a passo igual a gerada por um debugger e por isso pode confundir o seu funcionamento.

6.7.11 Empacotamento

Segundo (Fanglu Guo and Ferrie, 2008), empacotadores são programas que transformam a aparência de um executável sem afetar a sua execução semântica, criando com isso, novas variantes de malwares que conseguem fugir das ferramentas de detecção baseada em assinaturas. Em muitos casos, os escritores de malwares aplicam recursivamente diferentes combinações de múltiplos empacotadores em um mesmo malware, gerando várias cópias do malware aparentemente diferentes.

Como os anti-vírus tratam os empacotadores

Os anti-vírus tradicionais seguem os seguintes passos no tratamento de empacotadores:

1. Reconhecimento da família. Não é uma tarefa fácil, porque existem várias famílias e muitos empacotadores utilizam códigos polimórficos que alteram as aparências destas famílias.

2. Identificar sua versão. É primordial identificar a versão de um empacotador, porque dentro de uma mesma família existem rotinas diferentes de desempacotamento.

3. Criar um detector. Os dois passos acima são geralmente feitos de maneira manual. Este passo envolve a criação de um programa que, baseado nas informações coletadas, consegue detectar a família e a versão de um empacotador.

4. Criar um desempacotador. Diferente do detector, este programa traz o arquivo empacotado para a sua forma original, ou seja, realiza um desempacotamento.

Como funciona um empacotador

Para demonstrar o funcionamento de um empacotador, tomamos como exemplo o UPX (Ultimate Packer for eXecutables), que é um dos empacotadores mais utilizados por malwares.

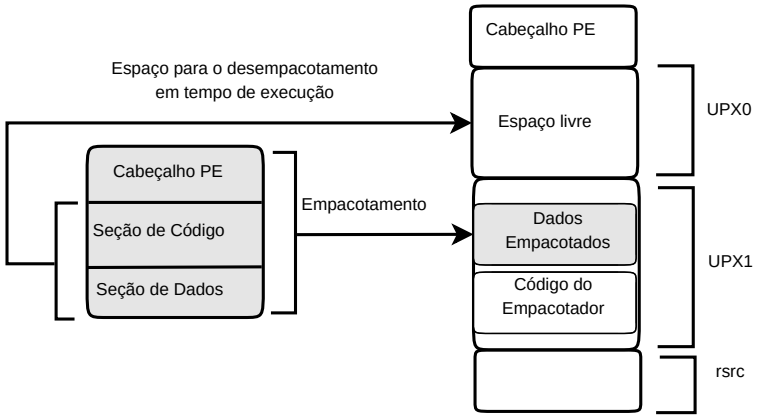

Figura 6.5 Empacotamento com o UPX

Quando o UPX comprime um binário PE, ele junta todas as suas seções, com exceção da seção de recursos, em uma simples seção. A seção de código e a seção de dados são comprimidas, empacotadas e armazenadas na área de dados da seção UPX1 do arquivo final.

O binário resultante tem três seções. A primeira seção, *UPX0*, é usada para reservar um intervalo de endereços que serão utilizados para restaurar o binário para a sua forma original em tempo de execução. A segunda seção, *UPX1*, contém os dados empacotados

seguidos imediatamente pelo código desempacotador. Para que o procedimento de desempacotamento funcione, é necessário que o ponto de entrada no cabeçalho do arquivo PE aponte para este código desempacotador. A terceira seção, *rsrc*, contém os dados de recursos, e também existe uma tabela de importação parcial que comporta todas as funções importadas de todas as DLLs (Dynamic Link Librarys)utilizadas pelo binário original.

Durante o procedimento de desempacotamento, o UPX realiza várias modificações nos atributos de escrita e execução das várias seções.

7 DigPE

7.1 A ferramenta

Conforme visto no capítulo 6, os escritores de malwares costumam utilizar-se de técnicas que tem o objetivo de proteger o código malicioso. Estas técnicas vão de simples encriptações, utilizando algoritmos já consagrados, passando pela utilização de funções da própria API do sistema operacional, chegando até a inserção direta no código da linguagem de montagem.

O software *DigPE*, é uma ferramenta projetada para trabalhar no estágio inicial de uma análise de código estática, ela tem como entrada um arquivo que segue o formato PE dos arquivos do sistema operacional Windows e a partir deste arquivo, extrairá informações úteis relativo ao próprio formato PE e também detectará através de assinaturas, a existência de empacotamento e de técnicas anti-debugging. Nas próximas seções serão discutidos o formato PE, a concepção do DigPE e alguns experimentos extraídos dos módulos que compõem a ferramenta.

7.2 O formato PE

O formato PE é o formato nativo para os arquivos do sistema operacional Windows, pertencentes a este formato, estão incluídos todos os arquivos executáveis, as DLLs, os arquivos .OCX, .CPL, a nova plataforma .NET e todos os drives que trabalham em modo kernel do sistema operacional. Abaixo teceremos comentários sobre os pontos mais relevantes deste formato de arquivo.

7.2.1 Estrutura básica

A estrutura mínima de um arquivo PE válido deve apresentar duas seções, uma seção para comportar o código e outra para os dados. Uma aplicação típica do Windows apresenta as seguintes seções:

Seção	Descrição
TEXT	Seção do código executável
DATA, RDATA OU BSS	Seção de dados
RSRC	Seção de recursos
EDATA	Seção de dados exportados
IDATA	Seção de dados importados
DEBUG	Seção de informações para o debug

Tabela 7.1 Tabela das Seções do Formato PE

Um ponto relevante neste formato é que o conteúdo do arquivo no disco será o mesmo quando este arquivo estiver carregado na memória. Mas, é importante notar que a localização de um item no disco diferirá de sua localização na memória, isto ocorre pelo fato de que, quando uma seção está na memória, ela estará alinhada pelo tamanho da página de memória e quando estiver no disco, estará alinhada pelo tamanho do bloco do disco. Diante deste comportamento, o formato PE deverá armazenar as informações referentes a ambos alinhamentos. Abaixo segue a estrutura básica do formato PE.

Figura 7.1 Estrutura Básica do Formato PE

Este formato segmentado habilita o processador a impor certas regras em como a memória será acessada. Seções são necessárias nos arquivos PE porque diferentes áreas no arquivo serão tratadas diferentemente pelo gerenciador de memória. Em tempo de execução, o gerenciador de memória configurará os direitos de acesso nas páginas de memórias das diferentes seções baseadas nas configurações encontradas em seus cabeçalhos. Isto determina se uma dada seção terá o acesso de leitura, escrita ou execução.

7.2.2 O cabeçalho DOS

Todos os arquivos PE começam com um cabeçalho DOS que ocupa os primeiros 64 bytes do arquivo. Este cabeçalho existe porque quando o programa esta sendo executado sob o sistema operacional DOS, este sistema pode reconhecer o arquivo como válido e executar o DOS stub, que fica armazenado imediatamente depois do cabeçalho. O DOS stub geralmente imprime a mensagem, *"Este programa deve ser executado sob o Microsoft Windows"*, mas pode conter qualquer código executável.

O cabeçalho DOS é uma estrutura definida no arquivo *windows.inc* ou no *winnt.h*, ela tem 19 membros, dos quais os campos *e_magic* e *e_lfanew* são de relevante interesse. O campo e_magic contém o valor *4Dh* e *5Ah*, significando as letras "MZ", que homenageia o arquiteto original do DOS, *Mark Zbikowsky*, e que também significa um cabeçalho DOS válido. O campo e_lfanew contém o offset do cabeçalho PE relativo ao começo do arquivo. Abaixo segue a estrutura do cabeçalho DOS.

```
IMAGE_DOS_HEADER  STRUCT
        e_magic         WORD    ?
        e_cblp          WORD    ?
        e_cp            WORD    ?
        e_crlc          WORD    ?
        e_cparhdr       WORD    ?
        e_minalloc      WORD    ?
        e_maxalloc      WORD    ?
        e_ss            WORD    ?
        e_sp            WORD    ?
        e_csum          WORD    ?
        e_ip            WORD    ?
        e_cs            WORD    ?
        e_lfarlc        WORD    ?
        e_ovno          WORD    ?
        e_res           WORD    4 dup(?)
```

```
        e_oemid                 WORD    ?
        e_oeminfo               WORD    ?
        e_res2                  WORD 10 dup(?)
        e_lfanew                WORD    ?
IMAGE_DOS_HEADER ENDS
```

Lista de Listagens 7.1 Estrutura IMAGE_DOS_HEADER

7.2.3 O cabeçalho PE

O cabeçalho PE é um termo geral para uma estrutura chamada IMAGE_NT_HEADERS. Esta estrutura contém informações essenciais utilizadas pelo *loader* do Windows. A estrutura é definida no arquivo *windows.inc* e possui os seguintes membros:

```
IMAGE_NT_HEADERS STRUCT
        Signature       WORD                    ?
        FileHeader      IMAGE_FILE_HEADER       <>
        OptionsHeader   IMAGE_OPTIONAL_HEADER32 <>
IMAGE_NT_HEADERS ENDS
```

Lista de Listagens 7.2 Estrutura IMAGE_NT_HEADERS

O campo *Signature* é uma DWORD que contém o valor *50h, 45h, 00h, 00h* significando as letras ”PE”, terminando com dois zeros. O campo *FileHeader*, contém informações sobre o layout físico e as propriedades do arquivo. O campo *OptionalHeader*, possui informações sobre o layout lógico do arquivo e seu tamanho é definido por um campo na estrutura IMAGE_FILE_HEADER.

```
IMAGE_FILE_HEADER STRUCT
        Machine                 WORD    ?
        NumberOfSections        WORD    ?
        TimeDateStamp           DWORD   ?
        PointerToSymbolTable    DWORD   ?
        NumberOfSymbols         DWORD   ?
        SizeOfOptionalHeader    WORD    ?
        Characteristics         WORD    ?
IMAGE_FILE_HEADER ENDS
```

Lista de Listagens 7.3 Estrutura IMAGE_FILE_HEADER

Dos campos mais relevantes da estrutura acima, podemos citar o campo *NumberOf-Sections*, que contém o número de seções do arquivo e o campo *Characteristics*, que indica se o arquivo PE é um executável ou uma DLL.

Com relação a estrutura IMAGE_OPTIONAL_HEADER32 mostrada abaixo, podemos destacar os seguintes campos:

```
IMAGE_OPTIONAL_HEADER32  STRUCT
    Magic                          WORD    ?
    MajorLinkerVersion             BYTE    ?
    MinorLinkerVersion             BYTE    ?
    SizeOfCode                     DWORD   ?
    SizeOfInitializedData          DWORD   ?
    SizeOfUnintializedData         DWORD   ?
    AddressOfEntryPoint            DWORD   ?
    BaseOfCode                     DWORD   ?
    BaseOfData                     DWORD   ?
    ImageBase                      DWORD   ?
    SectionAlignment               DWORD   ?
    FileAlignment                  DWORD   ?
    MajorOperatingSystemVersion    WORD    ?
    MinorOperationSystemVersion    WORD    ?
    MajorImageVersion              WORD    ?
    MinorImageVersion              WORD    ?
    MajorSubsystemVersion          WORD    ?
    MinorSubsystemVersion          WORD    ?
    Win32VersionValue              DWORD   ?
    SizeOfImage                    DOWRD   ?
    SizeOfHeaders                  DWORD   ?
    CheckSum                       DWORD   ?
    Subsystem                      WORD    ?
    DllCharacteristics             WORD    ?
    SizeOfStackReserve             DWORD   ?
    SizeOfStackCommit              DWORD   ?
    SizeOfHeapReserve              DWORD   ?
    SizeOfHeapCommit               DWORD   ?
    LoaderFlags                    DWORD   ?
    NumberOfRvaAndSizes            DWORD   ?
    DataDirectory                  IMAGE_DATA_DIRECTORY
```

```
IMAGE_OPTIONAL_HEADER32 ENDS
```

Lista de Listagens 7.4 Estrutura IMAGE_OPTIONAL_HEADER32

AddressOfEntryPoint: Este campo contém o RVA (Relative Virtual Address) da primeira instrução que será executada quando o loader estiver pronto para executar o arquivo. O campo AddressOfEntryPoint foi muito utilizado por escritores de malwares para apontar para o início de seus códigos maliciosos. Devido aos detectores baseados em heurísticas, esta técnica já não é tão utilizada.

ImageBase: Este campo contém o endereço desejado pelo loader para carregar o arquivo PE na memória. O termo ”desejado”, significa que nem sempre o loader do Windows consegue carregar o executável no endereço contido neste campo.

SectionAlignment: Representa a granularidade do alinhamento das seções na memória.

FileAlignment: Representa a granularidade do alinhamento das seções no disco.

SizeOfImage: Este campo contém o tamanho total do arquivo PE na memória. Ele contém o somatório de todos os cabeçalhos e todas as seções alinhadas pelo SectionAlignment.

SizeOfHeaders: Este campo contém o tamanho de todos os cabeçalhos adicionado ao tamanho da tabela de seção, seu conteúdo pode ser utilizado como o offset da primeira seção no arquivo PE.

DataDirectory: Contém uma array de 16 estruturas IMAGE_DATA_DIRECTORY, onde cada componente da array, relaciona-se com importantes áreas do arquivo PE.

7.2.4 A tabela de seção

Esta tabela se encontra logo abaixo do cabeçalho PE. Ela é composta de uma array de estruturas IMAGE_SECTION_HEADER e cada componente dessa array leva consigo informações sobre uma seção do arquivo PE. Esta estrutura é definida no arquivo *windows.inc* e é composta dos seguintes campos:

```
IMAGE_SECTION_HEADER STRUCT
  Name1   BYTE   IMAGE_SIZEOF_SHORT_NAME dup(?)
```

```
    union  Misc
        PhysicalAddress    DWORD   ?
        VirtualSize        DWORD   ?
    ends
    VirtualAddress         DWORD   ?
    SizeOfRawData          DWORD   ?
    PointerToRawData       DWORD   ?
    PointerToRelocations   DWORD   ?
    PointerToLinenumbers   DWORD   ?
    NumberOfRelocations    WORD    ?
    NumberOfLinenumbers    WORD    ?
    Characteristics        DWORD   ?
IMAGE_SECTION_HEADER  ENDS
```

Lista de Listagens 7.5 Estrutura IMAGE_SECTION_HEADER

Dos campos acima destacaremos:

Name1: Contém o nome da seção.

VirtualSize: Contém o tamanho real dos dados da seção em bytes.

VirtualAddress: Representa o RVA da seção, o loader do Windows utiliza este campo para mapear a seção na memória.

SizeOfRawData: Este campo contém o tamanho dos dados da seção no disco, arredondado pelo alinhamento de disco.

PointerToRawData: Representa um offset que aponta para o começo dos dados da seção. O loader utiliza este valor para encontrar os dados da seção no disco.

Characteristics: Contém um conjunto de flags que indica se a seção armazena código executável, se existem dados inicializados ou não inicializados e também se a seção pode ser escrita ou lida.

As seções encontram-se após a tabela de seção. No disco, cada seção começa em um offset que é múltiplo do valor encontrado no campo *FileAlignment* do OptionalHeader. Quando carregada na memória, as seções são alinhadas conforme o valor encontrado no campo *SectionAlignment* também do OptionalHeader.

7.2.5 As seções do arquivo PE

As seções contêm o principal conteúdo do arquivo e incluem códigos, dados, recursos e outras informações. Cada seção tem um cabeçalho e um corpo. Os cabeçalhos das seções estão contidos na tabela de seção e o corpo de cada seção é composto de uma estrutura rígida.

Uma aplicação do Windows tem tipicamente várias seções pré-definidas, algumas aplicações não necessitam de todas elas, enquanto outras podem definir seções mais específicas. As seções típicas de uma aplicação Windows serão descritas abaixo.

Seção de código: No Windows todos os segmentos de código residem em uma seção chamada *.text* ou *CODE*. Esta seção também contém o ponto de entrada do executável.

Seção de dados: Esta seção é representada pela seção *.bss*, na qual reside todos os dados não inicializados da aplicação, incluindo todas as variáveis definidas como estática, pela seção *.rdata*, que representa todos os dados de somente leitura, tal como, strings literais e constantes. Também faz parte da seção de dados, a seção *.data*, que contém todos os outros tipos de variáveis, com exceção das automáticas, que residem na pilha.

Seção de recursos: Representada pela seção *.rsrc*,contém todas as informações de todos os recursos pertencentes ao arquivo.

Seção de exportação: Representada pela seção *.edata*, contém o diretório de exportação de uma aplicação ou DLL. Quando presente, contém informações de todos os nomes e endereços das funções exportadas.

Seção de importação: Representada pela seção *.idata*, contém várias informações sobre as funções importadas, incluindo o diretório de importação e a tabela de endereços importados.

Debug: Esta seção é representada pela seção *.debug* e contém todas as informações relativas ao debug.

TLS: O sistema operacional Windows suporta múltiplas threads de execução por processos. Cada thread possui sua área de armazenamento privada para manter seus dados específicos. O linker pode criar uma seção chamada *.tls*, na qual define um layout para a área de armazenamento privado necessária para rotinas em arquivos executáveis e DLLs.

Seção de relocação: Representada pela seção *.reloc*, contém informações que possibilita ao loader realocar endereços em tempo de execução.

7.3 A concepção

A ferramenta DigPE foi construída utilizando como base a linguagem de programação python (Python Software Foundation, 2008), e um framework para extração de informações em arquivos PE disponível em (Ero Carrera, 2013). Ela foi concebida de forma escalável e modular e possui inicialmente módulos para a extração de informações do executável do Windows, detecção de técnicas de empacotamento e técnicas anti-debugging. Abaixo segue uma figura que mostra o fluxo lógico da ferramenta.

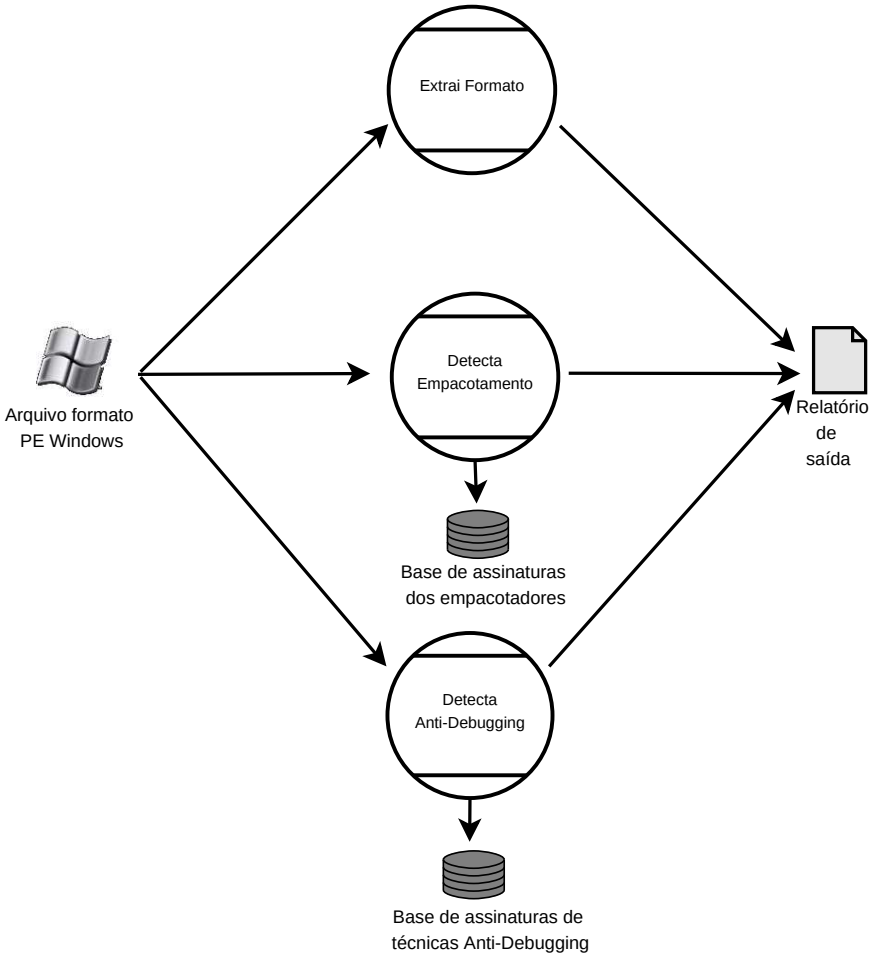

Figura 7.2 Fluxo Lógico do DigPE

7.3.1 Criação e otimização de assinaturas

O processo que envolve a criação das assinaturas que apontam a utilização de técnicas de autoproteção, é iniciado a partir dos códigos em linguagem assembly que representam tais técnicas. No primeiro momento, obtém-se a representação hexadecimal de todas as instruções, junto com seus operandos, que fazem parte do pedaço de código assembly. A sequência de códigos hexadecimal extraída, segue a sequência das instruções correspondentes. O segundo momento envolve a otimização das assinaturas, é analisada instrução por instrução, e verificada quais instruções ou operandos devem permanecer fixos e quais devem ser flexibilizados, como por exemplo, na instrução *push offset label1*, devemos tornar fixa a parte da assinatura relativa a instrução *push*, representada pelo valor hexadecimal 68, e a parte relativa ao endereço de memória *label1* deve ser flexibilizada, pois este endereço pode sofrer alterações quando o código for executado outras vezes. Voltando ao nosso exemplo de instrução, a assinatura final, considerando endereços de 32 bits, será igual a 68????????, onde a sequência ”????????” representando os 4 bytes do endereço de 32 bits, será ignorada pela rotina que pesquisa as assinaturas no binário sendo analisado.

7.3.2 O módulo de extração do formato PE

Este módulo escaneia todo o arquivo de formato PE do Windows extraindo informações das principais estruturas deste formato. As estruturas extraídas tais como, cabeçalho DOS (Disk Operating System), cabeçalho do arquivo, cabeçalho opcional, conteúdo sobre as seções, tabela de exportação e importação, fornecem informações valiosas para o analista que está investigando um determinado arquivo que segue este formato.

7.3.3 O módulo de detecção de empacotadores

O empacotamento é uma técnica muito utilizada pelos escritores de malwares no intuito de fugir da detecção dos anti-vírus. Este módulo é composto por uma base de **3520** assinaturas dos principais empacotadores atualmente existentes. Esta base de assinatura é uma base pública e está disponível em (Bob Soft, 2012). Durante o processo de análise neste módulo, a ferramenta navega de forma sequencial na base de assinaturas, extrai a informação de cada assinatura, calcula o seu tamanho, extrai uma quantidade de bytes do início da seção de código do arquivo de entrada equivalente ao tamanho da assinatura, e compara a informação da assinatura com a informação extraída do arquivo, se forem

iguais, é provável que o arquivo de entrada esteja utilizando algum método de empacotamento. A razão de extrair as informações de comparação apenas do início da seção de código é que, se existir algum método de empacotamento, a rotina de desempacotamento deste método será a primeira a ser executada e ela estará no início da seção de código do arquivo analisado.

7.3.4 O módulo de detecção de técnicas anti-debugging

Estas técnicas também são muito utilizadas por escritores de malwares e visam enganar a análise de códigos via debuggers. Este módulo é composto por uma base de **126** assinaturas, construída a partir de vários pedaços de códigos conseguidos através da Internet e em vários livros tal como (Sikorski and Horniq, 2012). Diferente da rotina de detecção de empacotadores, durante o processo de análise neste módulo, a ferramenta navega de forma sequencial na base de assinaturas, extrai a informação de cada assinatura, calcula o seu tamanho, e navega extraindo uma quantidade de bytes equivalente ao tamanho da assinatura deste o início até o final da seção de código do arquivo de entrada. De posse dos dados da assinatura e da extração do arquivo de entrada, o módulo compara estas informações, e se em algum momento elas forem iguais, é provável que o arquivo de entrada esteja utilizando alguma técnica de anti-debugging.

7.4 Exemplos de relatórios de saída da ferramenta DigPE

Vide apêndices A e B.

7.5 Experimentos

Também faz parte deste trabalho alguns experimentos realizados com os módulos de detecção de empacotamento e detecção de técnicas anti-debugging da ferramenta DigPE. Este experimento foi realizado com uma amostra de **17302** códigos maliciosos '.exe' e **1213** códigos maliciosos '.dll', todos 32 bits, conseguidos de forma devidamente autorizada do site (VirusSign, 2013).

7.5.1 Ambiente

Para processar a amostra dos códigos maliciosos, foi utilizado um simples PC HP Proliant ML 110, com 2G de memória RAM e 2 processadores Intel Pentium de 3.0 GHZ,

executando um sistema operacional Ubuntu 11.10 32 bits, com kernel versão 3.0.0-31-generic. Como linguagem de programação, foi utilizado o Python versão 2.7.

7.5.2 Experimento com empacotadores

A detecção de métodos de empacotamento é bem rápida, apenas verifica-se uma pequena quantidade de bytes no início da seção de código de cada arquivo analisado. Com a configuração da máquina descrita acima, o tempo gasto para processar todos os arquivos '.exe' foi de aproximadamente 1 hora, e para processar os arquivos '.dll', foram gastos algo em torno de 10 minutos. Como resultado deste processamento, gerou-se vários gráficos que demonstram os principais métodos de empacotamento utilizados pelos escritores de malwares. Seguem abaixo os gráficos que demonstram a frequência de utilização dos métodos de empacotamento nos dois tipos de arquivos analisados.

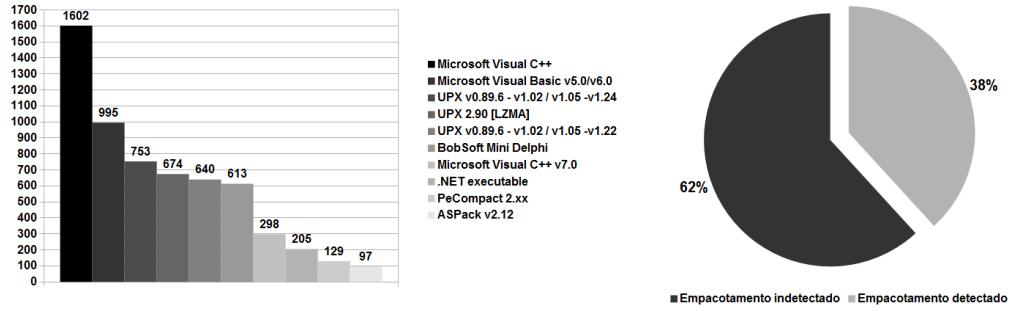

Figura 7.3 Frequência da utilização dos métodos de empacotamento em arquivos .exe

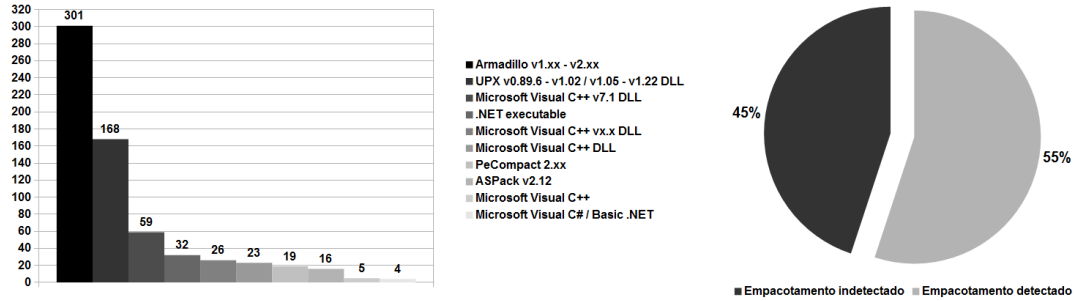

Figura 7.4 Frequência da utilização dos métodos de empacotamento em arquivos .dll

Os gráficos de barra acima mostram os 10 primeiros métodos de empacotamento mais utilizados, tanto em arquivos '.exe', quanto em arquivos '.dll'. Percebe-se através

dos gráficos de barra que, ignorando os empacotamentos utilizados pelas *ferramentas da Microsoft* nos arquivos '.exe' e pelo empacotador *Armadillo* nos arquivos '.dll', a família **UPX** ainda é um dos métodos mais utilizados pelos escritores de malwares para empacotar os seus códigos maliciosos, e isto é justificado por sua eficiência e usabilidade. Quanto aos valores dos gráficos de pizza, conclui-se que, o empacotamento como método de autoproteção de código, ainda é um artifício muito utilizado pelos escritores de malwares.

7.5.3 Experimento com anti-debugging

A detecção de métodos anti-debugging não é tão rápida, pois a assinatura é verificada por toda a seção de código de cada arquivo analisado. Com a configuração da máquina descrita acima, o tempo gasto para processar todos os arquivos '.exe' foi de aproximadamente 1 dia, 7 horas e 23 minutos e para processar os arquivos '.dll', foram gastas algo em torno de 5 horas. Na massa de dados analisada, foram encontrados poucos arquivos contendo algum tipo de técnica anti-debugging. Segue abaixo uma tabela com as informações dos arquivos e das técnicas anti-debugging encontradas.

Arquivo	Técnica Anti-Debugging	Descrição
virussign.com_0c5fa14188b 63810adf7ef21374a14e4.exe	IsDebuggerPresent	Detecção do debugger através do flag BeingDebugged da estrutura PEB
virussign.com_2d49c4dbb99 0725f150b8570e84abf54.exe	IsDebuggerPresent	Detecção do debugger através do flag BeingDebugged da estrutura PEB
virussign.com_17ad5cad7cf ce135750d3cc34b308226.exe	IsDebuggerPresent	Detecção do debugger através do flag BeingDebugged da estrutura PEB
virussign.com_8fd32b0e3a0 10a90186c3c0648c5776a.exe	IsDebuggerPresent	Detecção do debugger através do flag BeingDebugged da estrutura PEB
virussign.com_30c6b26a051 85b36fb7945b81ca6b10e.exe	Push/Pop Method BeingDebugged Flag	Detecção do debugger através do flag BeingDebugged da estrutura PEB utilizando as instruções Push/Pop
virussign.com_0b23bd34cdf 7d7fa6502e05d075ba269.exe	NtSetInformationThread	Detecção do debugger através da função API ntdll NtSetInformationThread() que seta o membro ThreadHideFromDebugger da classe ThreadInformationClass

Tabela 7.2 Técnicas anti-debugging

Apesar da pequena amostra de malwares apresentando técnicas anti-debugging, isto não implica que estas técnicas não sejam utilizadas por eles. O resultado da tabela acima, apenas significa que a base de assinaturas destas técnicas ainda é muito pequena. Se compararmos o tamanho da base de assinaturas dos empacotadores, que é de 3520 entradas,

com as 128 entradas da base de assinaturas das técnicas anti-debugging, notaremos que, para termos um resultado mais satisfatório, seria necessário um aumento significativo de entradas nesta base. Sem mencionar que, das 128 entradas da base de assinaturas das técnicas anti-debugging, 54 delas representam códigos de 64 bits, e o teste realizado envolveu apenas binários de 32 bits, ou seja, o universo de assinaturas consultadas foi ainda menor.

8 Conclusão

8.1 Considerações iniciais

Neste capítulo são apresentadas as conclusões deste trabalho. Na seção 8.2, são explanadas as principais contribuições fornecidas por esta dissertação e na seção 8.4, são apresentadas algumas ideias que podem contribuir para a realização de trabalhos futuros.

8.2 Principais contribuições

As palavras de Sun Tzu encontradas na epígrafe deste trabalho, representam de maneira perfeita o quanto é necessária a investigação dos códigos maliciosos para que se aplique as corretas ações mitigatórias, como uma melhor otimização da base de assinaturas dos malwares, com o objetivo de combater os efeitos danosos causados por estes tipos de códigos. Conforme relatório de ameaças do quarto trimestre de 2012 da McAfee Labs encontrado em (McAfee Labs, 2012), a quantidade de novos malwares cresce com grande velocidade, e para que se crie ferramentas automatizadas visando atacar esse crescimento, é necessário, antes de tudo, que se tenha um bom conhecimento destes códigos. Sabe-se ainda que grande parte destes novos malwares são apenas maneiras diferentes de escrever os velhos códigos, e se estes códigos antigos já foram analisados anteriormente, será mais fácil encontrar vacinas para a nova mutação.

Este trabalho traz contribuições a todas as pessoas que desejam conhecer com uma maior profundidade o universo da análise de malware. No capítulo que fala sobre as técnicas de análises, foram abordadas as duas grandes técnicas de análise de código, a análise estática e a análise dinâmica. Neste capítulo foram mostradas as diferentes metodologias de análises realizadas quando um código está em repouso ou em plena

execução. No capítulo sobre a classificação de malwares, mostramos as diferentes maneiras de tentar classificar um determinado código, as dificuldades de classificação por causa das constantes sobreposições de características e os diferentes tipos de malwares. No capítulo sobre as estratégias de infecção, apresentamos as diferentes maneiras utilizadas pelos vírus e worms para infectar e proliferar a sua espécie. O capítulo que fala sobre as estratégias de autoproteção é um dos pontos principais deste trabalho, foi através do estudo destas técnicas de autoproteção que se idealizou uma ferramenta capaz de identificar algumas delas e foi esta ferramenta que originou o capítulo seguinte, onde se tece comentários sobre o objetivo, a concepção e alguns experimentos utilizando este software.

8.3 Trabalhos relacionados

Como trabalhos relacionados ao processo de análise de malwares, poderemos mencionar (Tzu-Yen Wang and Hsieh, 2009) e (Kolter and Maloof, 2004), que utilizaram técnicas de aprendizagem de máquinas para identificar códigos maliciosos e (Kil Jin Brandini Park, 2011) que utiliza cálculos estatísticos para o reconhecimento de empacotamento de executáveis.

8.4 Trabalhos futuros

Existe muito o que se pesquisar dentro do processo que envolve a análise de malware, como uma ideia de trabalho futuro, poderemos considerar a ampliação da ferramenta construída estudando novas técnicas de autoproteção tais como, anti-disassembler e anti-emulação. Para a detecção destas novas técnicas, deveremos pesquisar códigos que realizam estas proteções e fazer com que estes códigos virem assinaturas para uma posterior análise de código estática. Também é idealizada uma mudança radical na maneira de detecção destas técnicas de autoproteção, em vez de se utilizar assinaturas, poderemos utilizar técnicas de aprendizagem de máquina para analisar o código em execução e identificar se estes códigos possuem alguma característica de autoproteção.

Referências

Aycock, J. (2006). *Computer Viruses and Malware*. Springer.

Ball, T. (1999). The concept of dynamic analysis. pages 216–234.

Bob Soft (2012). Base de assinaturas de empacotadores. http://www.PEiD.info /BobSoft/Downloads.html. Acessado em Jun/2013.

Cliff Wang, Mihai Christodorescu, S. J. D. M. and Song, D. (2007). *Malware Detection*. Springer.

Cohen (1984). Experiments with Computer Viruses. http://www.all.net /books/virus/part5.html. Acessado em Jun/2012.

Cohen, F. (1994). *A Short Course on Computer Viruses*. Wiley; 2 edition.

Eilam, E. (2005). *Reversing: Secrets of Reverse Engineering*. Wiley Publishing.

Erickson, J. (2008). *Hacking:The Art of Exploration*. No Starch Press; 2 edition.

Ero Carrera (2013). Python module to read and work with PE (Portable Executable) files. https://code.google.com/p/pefile/. Acessado em Jun/2013.

Fanglu Guo, T.-c. C. and Ferrie, P. (2008). A study of the packer problem and its solutions. pages 1–18.

Goppit (2006). *Portable Executable File Format - A Reverse Enginner View*. CodeBreakers Magazine.

Irvine, K. R. (2007). *Assembly Language For X86 Processors*. Prentice Hall.

Justin Ferguson, Dan Kaminsky, J. L. L. M. and Pearce, W. (2008). *Reverse Engineering Code with IDA Pro*. Syngress.

Kil Jin Brandini Park, Rodrigo Ruiz, A. M. (2011). Binstat:ferramenta para reconhecimento de executáveis empacotados.

Kolter, J. and Maloof, M. (2004). Learning to detect malicious executables in the wild. pages 470–478.

Manzano, J. A. N. G. (2007). *Fundamentos em Programação Assembly*. Erica.

McAfee Labs (2012). McAfee Threats Report:Fourth Quarter 2012. http://www.mcafee.com/br/resources/reports/rp-quarterly-threat-q4-2012.pdf ?cid=BHP012. Acessado em Abril/2013.

Michael Hale Ligh, Steven Adair, B. H. and Richard, M. (2011). *Malware Analyst's Cookbook and DVD, Tools and Techniques for Fighting Malicious Code*. Wiley Publishing.

Python Software Foundation (2008). Python Programming Language. http://www.python.org. Acessado em Jun/2013.

Robert Fitzgerald, Todd B. Knoblock, B. S. a. D. T. (1999). Marmot: An optimizing compiler for java.

Sikorski, M. and Horniq, A. (2012). *Pratical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press.

Singh, A. (2009). *Identifying Malicious Code Through Reverse Engeneering*. Springer.

Snowplow (2012). The history of worm like programs. http://www.snowplow.org /tom/worm/history.html. Acessado em Jun/2012.

Stallings, W. (2010). *Criptografia e Segurança de Redes*. Person; 4 edição.

Szor, P. (2005). *Virus Research and Defense*. Symantec Press.

Tzu-Yen Wang, C.-H. W. and Hsieh, C. C. (2009). Detecting unknown malicious executables using portable executables headers. pages 278–284.

VirusSign (2013). VirusSign | Free Virus Samples, Malware Samples Free Download. www.virussign.com. Acessado em Mai/2013.

Apêndices

A

Relatório de um arquivo empacotado

```
DigPE Versao 0.1.0
Data: 22/04/2013  Hora: 11:11:21
Arq : virussign.com_b4e45ca7a17787b4d8c4391897025102.exe
MD5 : b4e45ca7a17787b4d8c4391897025102
SHA1: dd41698b181f05b48c461f31bff5a39f240bdcd4
```

Estrutura do formato Portable Executable:

```
---------- DOS Header ----------
e_magic     : 0x5a4d
e_cblp      : 0x90
e_cp        : 0x3
e_crlc      : 0x0
e_cparhdr   : 0x4
e_minalloc  : 0x0
e_maxalloc  : 0xffff
e_ss        : 0x0
e_sp        : 0xb8
e_csum      : 0x0
e_ip        : 0x0
e_cs        : 0x0
e_lfarlc    : 0x40
e_ovno      : 0x0
e_res       :
e_oemid     : 0x0
```

```
e_oeminfo   : 0x0
e_res2      :
e_lfanew    : 0x80

---------------------- NT Header ----------------------
Signature : 0x4550

---------------------- File Header ----------------------
Machine                 : 0x14c
NumberOfSections        : 0x3
TimeDateStamp           : 0x3e37849b
PointerToSymbolTable    : 0x0
NumberOfSymbols         : 0x0
SizeOfOptionalHeader    : 0xe0
Characteristics         : 0x10f

---------------------- Optional Header ----------------------
Magic                           : 0x10b
MajorLinkerVersion              : 0x1
MinorLinkerVersion              : 0x3
SizeOfCode                      : 0x1000
SizeOfInitializedData           : 0x1000
SizeOfUninitializedData         : 0x5000
AddressOfEntryPoint             : 0x6820
BaseOfCode                      : 0x6000
BaseOfData                      : 0x7000
ImageBase                       : 0x400000
SectionAlignment                : 0x1000
FileAlignment                   : 0x200
MajorOperatingSystemVersion     : 0x1
MinorOperatingSystemVersion     : 0x0
MajorImageVersion               : 0x0
MinorImageVersion               : 0x0
MajorSubsystemVersion           : 0x4
MinorSubsystemVersion           : 0x0
Reserved1                       : 0x0
SizeOfImage                     : 0x8000
SizeOfHeaders                   : 0x1000
```

```
CheckSum                    : 0x0
Subsystem                   : 0x2
DllCharacteristics          : 0x0
SizeOfStackReserve          : 0x100000
SizeOfStackCommit           : 0x1000
SizeOfHeapReserve           : 0x100000
SizeOfHeapCommit            : 0x1000
LoaderFlags                 : 0x0
NumberOfRvaAndSizes         : 0x10

---------------------- Section Header------------------

Name                        : UPX0
Misc                        : 0x5000
Misc_PhysicalAddress        : 0x5000
VirtualAddress              : 0x1000
Misc_VirtualSize            : 0x5000
SizeOfRawData               : 0x0
PointerToRawData            : 0x200
PointerToRelocations        : 0x0
PointerToLinenumbers        : 0x0
NumberOfRelocations         : 0x0
NumberOfLinenumbers         : 0x0
Characteristics             : 0xe0000080

Name                        : UPX1
Misc                        : 0x1000
Misc_PhysicalAddress        : 0x1000
VirtualAddress              : 0x6000
Misc_VirtualSize            : 0x1000
SizeOfRawData               : 0xa00
PointerToRawData            : 0x200
PointerToRelocations        : 0x0
PointerToLinenumbers        : 0x0
NumberOfRelocations         : 0x0
NumberOfLinenumbers         : 0x0
Characteristics             : 0xe0000040

Name                        : .rsrc
```

```
Misc                    : 0x1000
Misc_PhysicalAddress    : 0x1000
VirtualAddress          : 0x7000
Misc_VirtualSize        : 0x1000
SizeOfRawData           : 0x200
PointerToRawData        : 0xc00
PointerToRelocations    : 0x0
PointerToLinenumbers    : 0x0
NumberOfRelocations     : 0x0
NumberOfLinenumbers     : 0x0
Characteristics         : 0xc0000040

------------------- Import Directory -------------------
[ KERNEL32.DLL ]
Characteristics     : 0x0
FirstThunk          : 0x70a8
ForwarderChain      : 0x0
Name                : 0x70c8
OriginalFirstThunk  : 0x0
TimeDateStamp       : 0x0
------------------- Import Functions -------------------
address             : 0x4070a8
bound               : None
hint                : 0x0
hint_name_table_rva : 0x70ea
import_by_ordinal   : False
name                : LoadLibraryA
ordinal             : None
ordinal_offset      : 0xca8
thunk_offset        : 0xca8
thunk_rva           : 0x70a8

address             : 0x4070ac
bound               : None
hint                : 0x0
hint_name_table_rva : 0x70f8
import_by_ordinal   : False
name                : GetProcAddress
```

```
ordinal              : None
ordinal_offset       : 0xcac
thunk_offset         : 0xcac
thunk_rva            : 0x70ac

address              : 0x4070b0
bound                : None
hint                 : 0x0
hint_name_table_rva  : 0x7108
import_by_ordinal    : False
name                 : ExitProcess
ordinal              : None
ordinal_offset       : 0xcb0
thunk_offset         : 0xcb0
thunk_rva            : 0x70b0

[  CRTDLL.DLL  ]
Characteristics      : 0x0
FirstThunk           : 0x70b8
ForwarderChain       : 0x0
Name                 : 0x70d5
OriginalFirstThunk   : 0x0
TimeDateStamp        : 0x0
———————————— Import Functions ————————————
address              : 0x4070b8
bound                : None
hint                 : 0x0
hint_name_table_rva  : 0x7116
import_by_ordinal    : False
name                 : exit
ordinal              : None
ordinal_offset       : 0xcb8
thunk_offset         : 0xcb8
thunk_rva            : 0x70b8

[  USER32.DLL  ]
Characteristics      : 0x0
FirstThunk           : 0x70c0
```

```
ForwarderChain      : 0x0
Name                : 0x70e0
OriginalFirstThunk  : 0x0
TimeDateStamp       : 0x0
--------------------- Import Functions ---------------
address             : 0x4070c0
bound               : None
hint                : 0x0
hint_name_table_rva : 0x711c
import_by_ordinal   : False
name                : EndDialog
ordinal             : None
ordinal_offset      : 0xcc0
thunk_offset        : 0xcc0
thunk_rva           : 0x70c0


------------------------------------------------------------

Tecnicas de Auto-Protecao:

[ Empacotamento ]
UPX 2.90 [LZMA] -> Markus Oberhumer, L. Molnar & J.
    Reiser

[ Anti-Debugging ]
Offset: - Descricao: Anti-Debugging indetectado
```

B

Relatório de um arquivo com anti-debugging

```
DigPE Versao 0.1.0
Data: 22/04/2013  Hora: 09:45:13
Arq : virussign.com_17ad5cad7cfce135750d3cc34b308226.exe
MD5 : 17ad5cad7cfce135750d3cc34b308226
SHA1: 89d8bc4597489a30510b93b6773b9e28ee1e4787
```

Estrutura do formato Portable Executable:

```
---------- DOS Header ----------
e_magic     : 0x5a4d
e_cblp      : 0x90
e_cp        : 0x3
e_crlc      : 0x0
e_cparhdr   : 0x4
e_minalloc  : 0x0
e_maxalloc  : 0xffff
e_ss        : 0x0
e_sp        : 0xb8
e_csum      : 0x0
e_ip        : 0x0
e_cs        : 0x0
e_lfarlc    : 0x40
e_ovno      : 0x0
```

```
e_res      :
e_oemid    : 0x0
e_oeminfo  : 0x0
e_res2     :
e_lfanew   : 0x80

──────────────────── NT Header ────────────────────
Signature : 0x4550

──────────────────── File Header ──────────────────
Machine              : 0x14c
NumberOfSections     : 0x4
TimeDateStamp        : 0x4eb8440d
PointerToSymbolTable : 0x0
NumberOfSymbols      : 0x0
SizeOfOptionalHeader : 0xe0
Characteristics      : 0x10f

──────────────────── Optional Header ──────────────
Magic                       : 0x10b
MajorLinkerVersion          : 0x5
MinorLinkerVersion          : 0xc
SizeOfCode                  : 0xe00
SizeOfInitializedData       : 0x2c800
SizeOfUninitializedData     : 0x0
AddressOfEntryPoint         : 0x1ad8
BaseOfCode                  : 0x1000
BaseOfData                  : 0x2000
ImageBase                   : 0x400000
SectionAlignment            : 0x1000
FileAlignment               : 0x200
MajorOperatingSystemVersion : 0x4
MinorOperatingSystemVersion : 0x0
MajorImageVersion           : 0x0
MinorImageVersion           : 0x0
MajorSubsystemVersion       : 0x4
MinorSubsystemVersion       : 0x0
Reserved1                   : 0x0
```

```
SizeOfImage                 : 0x3d000
SizeOfHeaders               : 0x400
CheckSum                    : 0x0
Subsystem                   : 0x2
DllCharacteristics          : 0x0
SizeOfStackReserve          : 0x100000
SizeOfStackCommit           : 0x1000
SizeOfHeapReserve           : 0x100000
SizeOfHeapCommit            : 0x1000
LoaderFlags                 : 0x0
NumberOfRvaAndSizes         : 0x10

——————————————— Section Header———————————————
Name                        : .text
Misc                        : 0xc1c
Misc_PhysicalAddress        : 0xc1c
VirtualAddress              : 0x1000
Misc_VirtualSize            : 0xc1c
SizeOfRawData               : 0xe00
PointerToRawData            : 0x400
PointerToRelocations        : 0x0
PointerToLinenumbers        : 0x0
NumberOfRelocations         : 0x0
NumberOfLinenumbers         : 0x0
Characteristics             : 0x60000020

Name                        : .rdata
Misc                        : 0x4c0
Misc_PhysicalAddress        : 0x4c0
VirtualAddress              : 0x2000
Misc_VirtualSize            : 0x4c0
SizeOfRawData               : 0x600
PointerToRawData            : 0x1200
PointerToRelocations        : 0x0
PointerToLinenumbers        : 0x0
NumberOfRelocations         : 0x0
NumberOfLinenumbers         : 0x0
Characteristics             : 0x40000040
```

```
Name                    : .data
Misc                    : 0xd6f0
Misc_PhysicalAddress    : 0xd6f0
VirtualAddress          : 0x3000
Misc_VirtualSize        : 0xd6f0
SizeOfRawData           : 0x600
PointerToRawData        : 0x1800
PointerToRelocations    : 0x0
PointerToLinenumbers    : 0x0
NumberOfRelocations     : 0x0
NumberOfLinenumbers     : 0x0
Characteristics         : 0xc0000040

Name                    : .rsrc
Misc                    : 0x2ba8c
Misc_PhysicalAddress    : 0x2ba8c
VirtualAddress          : 0x11000
Misc_VirtualSize        : 0x2ba8c
SizeOfRawData           : 0x2bc00
PointerToRawData        : 0x1e00
PointerToRelocations    : 0x0
PointerToLinenumbers    : 0x0
NumberOfRelocations     : 0x0
NumberOfLinenumbers     : 0x0
Characteristics         : 0xc0000040

------------------- Import Directory -------------------
[   user32.dll   ]
Characteristics     : 0x2190
FirstThunk          : 0x2084
ForwarderChain      : 0x0
Name                : 0x229e
OriginalFirstThunk  : 0x2190
TimeDateStamp       : 0x0
------------------- Import Functions -------------------
address             : 0x402084
bound               : None
```

```
hint                : 0x26a
hint_name_table_rva : 0x228e
import_by_ordinal   : False
name                : UpdateWindow
ordinal             : None
ordinal_offset      : 0x1390
thunk_offset        : 0x1390
thunk_rva           : 0x2190

address             : 0x402088
bound               : None
hint                : 0x25e
hint_name_table_rva : 0x227a
import_by_ordinal   : False
name                : TranslateMessage
ordinal             : None
ordinal_offset      : 0x1394
thunk_offset        : 0x1394
thunk_rva           : 0x2194

address             : 0x40208c
bound               : None
hint                : 0x248
hint_name_table_rva : 0x226c
import_by_ordinal   : False
name                : ShowWindow
ordinal             : None
ordinal_offset      : 0x1398
thunk_offset        : 0x1398
thunk_rva           : 0x2198

address             : 0x402090
bound               : None
hint                : 0x1fd
hint_name_table_rva : 0x225c
import_by_ordinal   : False
name                : SendMessageA
ordinal             : None
```

```
ordinal_offset      : 0x139c
thunk_offset        : 0x139c
thunk_rva           : 0x219c

address             : 0x402094
bound               : None
hint                : 0x1e1
hint_name_table_rva : 0x2248
import_by_ordinal   : False
name                : RegisterClassExA
ordinal             : None
ordinal_offset      : 0x13a0
thunk_offset        : 0x13a0
thunk_rva           : 0x21a0

address             : 0x402098
bound               : None
hint                : 0x1d5
hint_name_table_rva : 0x2236
import_by_ordinal   : False
name                : PostQuitMessage
ordinal             : None
ordinal_offset      : 0x13a4
thunk_offset        : 0x13a4
thunk_rva           : 0x21a4

address             : 0x40209c
bound               : None
hint                : 0x1b1
hint_name_table_rva : 0x2228
import_by_ordinal   : False
name                : MessageBoxA
ordinal             : None
ordinal_offset      : 0x13a8
thunk_offset        : 0x13a8
thunk_rva           : 0x21a8

address             : 0x4020a0
```

```
bound                : None
hint                 : 0x198
hint_name_table_rva  : 0x221c
import_by_ordinal    : False
name                 : LoadIconA
ordinal              : None
ordinal_offset       : 0x13ac
thunk_offset         : 0x13ac
thunk_rva            : 0x21ac

address              : 0x4020a4
bound                : None
hint                 : 0x194
hint_name_table_rva  : 0x220e
import_by_ordinal    : False
name                 : LoadCursorA
ordinal              : None
ordinal_offset       : 0x13b0
thunk_offset         : 0x13b0
thunk_rva            : 0x21b0

address              : 0x4020a8
bound                : None
hint                 : 0x122
hint_name_table_rva  : 0x2200
import_by_ordinal    : False
name                 : GetMessageA
ordinal              : None
ordinal_offset       : 0x13b4
thunk_offset         : 0x13b4
thunk_rva            : 0x21b4

address              : 0x4020ac
bound                : None
hint                 : 0x93
hint_name_table_rva  : 0x21ec
import_by_ordinal    : False
name                 : DispatchMessageA
```

```
ordinal              : None
ordinal_offset       : 0x13b8
thunk_offset         : 0x13b8
thunk_rva            : 0x21b8

address              : 0x4020b0
bound                : None
hint                 : 0x83
hint_name_table_rva  : 0x21da
import_by_ordinal    : False
name                 : DefWindowProcA
ordinal              : None
ordinal_offset       : 0x13bc
thunk_offset         : 0x13bc
thunk_rva            : 0x21bc

address              : 0x4020b4
bound                : None
hint                 : 0x56
hint_name_table_rva  : 0x21c8
import_by_ordinal    : False
name                 : CreateWindowExA
ordinal              : None
ordinal_offset       : 0x13c0
thunk_offset         : 0x13c0
thunk_rva            : 0x21c0

[  kernel32.dll  ]
Characteristics    : 0x210c
FirstThunk         : 0x2000
ForwarderChain     : 0x0
Name               : 0x2492
OriginalFirstThunk : 0x210c
TimeDateStamp      : 0x0
------------------- Import Functions ----------------
address              : 0x402000
bound                : None
hint                 : 0x134
```

```
hint_name_table_rva  : 0x2344
import_by_ordinal    : False
name                 : GetModuleHandleA
ordinal              : None
ordinal_offset       : 0x130c
thunk_offset         : 0x130c
thunk_rva            : 0x210c

address              : 0x402004
bound                : None
hint                 : 0x1bd
hint_name_table_rva  : 0x23d6
import_by_ordinal    : False
name                 : HeapAlloc
ordinal              : None
ordinal_offset       : 0x1310
thunk_offset         : 0x1310
thunk_rva            : 0x2110

address              : 0x402008
bound                : None
hint                 : 0x31d
hint_name_table_rva  : 0x2486
import_by_ordinal    : False
name                 : lstrlenA
ordinal              : None
ordinal_offset       : 0x1314
thunk_offset         : 0x1314
thunk_rva            : 0x2114

address              : 0x40200c
bound                : None
hint                 : 0x31b
hint_name_table_rva  : 0x247a
import_by_ordinal    : False
name                 : lstrcpynA
ordinal              : None
ordinal_offset       : 0x1318
```

```
thunk_offset          : 0x1318
thunk_rva             : 0x2118

address               : 0x402010
bound                 : None
hint                  : 0x319
hint_name_table_rva   : 0x246e
import_by_ordinal     : False
name                  : lstrcpyA
ordinal               : None
ordinal_offset        : 0x131c
thunk_offset          : 0x131c
thunk_rva             : 0x211c

address               : 0x402014
bound                 : None
hint                  : 0x313
hint_name_table_rva   : 0x2462
import_by_ordinal     : False
name                  : lstrcatA
ordinal               : None
ordinal_offset        : 0x1320
thunk_offset          : 0x1320
thunk_rva             : 0x2120

address               : 0x402018
bound                 : None
hint                  : 0x2fb
hint_name_table_rva   : 0x2456
import_by_ordinal     : False
name                  : WriteFile
ordinal               : None
ordinal_offset        : 0x1324
thunk_offset          : 0x1324
thunk_rva             : 0x2124

address               : 0x40201c
bound                 : None
```

```
hint                : 0x2ba
hint_name_table_rva : 0x2444
import_by_ordinal   : False
name                : SizeofResource
ordinal             : None
ordinal_offset      : 0x1328
thunk_offset        : 0x1328
thunk_rva           : 0x2128

address             : 0x402020
bound               : None
hint                : 0x287
hint_name_table_rva : 0x242e
import_by_ordinal   : False
name                : SetFileAttributesA
ordinal             : None
ordinal_offset      : 0x132c
thunk_offset        : 0x132c
thunk_rva           : 0x212c

address             : 0x402024
bound               : None
hint                : 0x25a
hint_name_table_rva : 0x241e
import_by_ordinal   : False
name                : RtlMoveMemory
ordinal             : None
ordinal_offset      : 0x1330
thunk_offset        : 0x1330
thunk_rva           : 0x2130

address             : 0x402028
bound               : None
hint                : 0x1fd
hint_name_table_rva : 0x240e
import_by_ordinal   : False
name                : LockResource
ordinal             : None
```

```
ordinal_offset      : 0x1334
thunk_offset        : 0x1334
thunk_rva           : 0x2134

address             : 0x40202c
bound               : None
hint                : 0x1ef
hint_name_table_rva : 0x23fe
import_by_ordinal   : False
name                : LoadResource
ordinal             : None
ordinal_offset      : 0x1338
thunk_offset        : 0x1338
thunk_rva           : 0x2138

address             : 0x402030
bound               : None
hint                : 0x1ea
hint_name_table_rva : 0x23ee
import_by_ordinal   : False
name                : LoadLibraryA
ordinal             : None
ordinal_offset      : 0x133c
thunk_offset        : 0x133c
thunk_rva           : 0x213c

address             : 0x402034
bound               : None
hint                : 0x23
hint_name_table_rva : 0x22aa
import_by_ordinal   : False
name                : CloseHandle
ordinal             : None
ordinal_offset      : 0x1340
thunk_offset        : 0x1340
thunk_rva           : 0x2140

address             : 0x402038
```

```
bound                : None
hint                 : 0x3d
hint_name_table_rva  : 0x22b8
import_by_ordinal    : False
name                 : CreateFileA
ordinal              : None
ordinal_offset       : 0x1344
thunk_offset         : 0x1344
thunk_rva            : 0x2144

address              : 0x40203c
bound                : None
hint                 : 0x9b
hint_name_table_rva  : 0x22c6
import_by_ordinal    : False
name                 : ExitProcess
ordinal              : None
ordinal_offset       : 0x1348
thunk_offset         : 0x1348
thunk_rva            : 0x2148

address              : 0x402040
bound                : None
hint                 : 0xc0
hint_name_table_rva  : 0x22d4
import_by_ordinal    : False
name                 : FindResourceA
ordinal              : None
ordinal_offset       : 0x134c
thunk_offset         : 0x134c
thunk_rva            : 0x214c

address              : 0x402044
bound                : None
hint                 : 0xd3
hint_name_table_rva  : 0x22e4
import_by_ordinal    : False
name                 : FreeResource
```

```
ordinal              : None
ordinal_offset       : 0x1350
thunk_offset         : 0x1350
thunk_rva            : 0x2150

address              : 0x402048
bound                : None
hint                 : 0xe6
hint_name_table_rva  : 0x22f4
import_by_ordinal    : False
name                 : GetCommandLineA
ordinal              : None
ordinal_offset       : 0x1354
thunk_offset         : 0x1354
thunk_rva            : 0x2154

address              : 0x40204c
bound                : None
hint                 : 0x113
hint_name_table_rva  : 0x2306
import_by_ordinal    : False
name                 : GetEnvironmentVariableA
ordinal              : None
ordinal_offset       : 0x1358
thunk_offset         : 0x1358
thunk_rva            : 0x2158

address              : 0x402050
bound                : None
hint                 : 0x11c
hint_name_table_rva  : 0x2320
import_by_ordinal    : False
name                 : GetFileSize
ordinal              : None
ordinal_offset       : 0x135c
thunk_offset         : 0x135c
thunk_rva            : 0x215c
```

```
address             : 0x402054
bound               : None
hint                : 0x132
hint_name_table_rva : 0x232e
import_by_ordinal   : False
name                : GetModuleFileNameA
ordinal             : None
ordinal_offset      : 0x1360
thunk_offset        : 0x1360
thunk_rva           : 0x2160

address             : 0x402058
bound               : None
hint                : 0x1ac
hint_name_table_rva : 0x23c8
import_by_ordinal   : False
name                : GlobalFree
ordinal             : None
ordinal_offset      : 0x1364
thunk_offset        : 0x1364
thunk_rva           : 0x2164

address             : 0x40205c
bound               : None
hint                : 0x153
hint_name_table_rva : 0x2358
import_by_ordinal   : False
name                : GetProcAddress
ordinal             : None
ordinal_offset      : 0x1368
thunk_offset        : 0x1368
thunk_rva           : 0x2168

address             : 0x402060
bound               : None
hint                : 0x156
hint_name_table_rva : 0x236a
import_by_ordinal   : False
```

```
name                : GetProcessHeap
ordinal             : None
ordinal_offset      : 0x136c
thunk_offset        : 0x136c
thunk_rva           : 0x216c

address             : 0x402064
bound               : None
hint                : 0x172
hint_name_table_rva : 0x237c
import_by_ordinal   : False
name                : GetSystemDirectoryA
ordinal             : None
ordinal_offset      : 0x1370
thunk_offset        : 0x1370
thunk_rva           : 0x2170

address             : 0x402068
bound               : None
hint                : 0x184
hint_name_table_rva : 0x2392
import_by_ordinal   : False
name                : GetTempPathA
ordinal             : None
ordinal_offset      : 0x1374
thunk_offset        : 0x1374
thunk_rva           : 0x2174

address             : 0x40206c
bound               : None
hint                : 0x1a0
hint_name_table_rva : 0x23a2
import_by_ordinal   : False
name                : GetWindowsDirectoryA
ordinal             : None
ordinal_offset      : 0x1378
thunk_offset        : 0x1378
thunk_rva           : 0x2178
```

```
address              : 0x402070
bound                : None
hint                 : 0x1a5
hint_name_table_rva  : 0x23ba
import_by_ordinal    : False
name                 : GlobalAlloc
ordinal              : None
ordinal_offset       : 0x137c
thunk_offset         : 0x137c
thunk_rva            : 0x217c

address              : 0x402074
bound                : None
hint                 : 0x1c1
hint_name_table_rva  : 0x23e2
import_by_ordinal    : False
name                 : HeapFree
ordinal              : None
ordinal_offset       : 0x1380
thunk_offset         : 0x1380
thunk_rva            : 0x2180

[  shlwapi.dll  ]
Characteristics     : 0x2188
FirstThunk          : 0x207c
ForwarderChain      : 0x0
Name                : 0x24b4
OriginalFirstThunk  : 0x2188
TimeDateStamp       : 0x0
———————————————— Import Functions ————————————————
address              : 0x40207c
bound                : None
hint                 : 0x2e
hint_name_table_rva  : 0x24a0
import_by_ordinal    : False
name                 : PathFindFileNameA
ordinal              : None
```

```
ordinal_offset       : 0x1388
thunk_offset         : 0x1388
thunk_rva            : 0x2188
```

```
Tecnicas de Auto-Protecao:

[ Empacotamento ]
Empacotamento indetectado

[ Anti-Debugging ]
Offset: 1367 Descricao: IsDebuggerPresent 32bit on
   Operating System 32bit
```