UNIVERSIDADE ESTADUAL PAULISTA

Faculdade de Ciências - Bauru

Bacharelado em Ciência da Computação

Gustavo André Arrabal de Souza

PROPOSTA DE ARQUITETURA E PROTOCOLO PARA A GERAÇÃO AUTOMÁTICA DE ASSINATURAS DE *MALWARES*

UNESP

2012

Gustavo André Arrabal de Souza

PROPOSTA DE ARQUITETURA E PROTOCOLO PARA A GERAÇÃO AUTOMÁTICA DE ASSINATURAS DE *MALWARES*

Orientador: Marcos Antônio Cavenaghi

Monografia apresentada junto à disciplina Projeto e Implementação de Sistemas II, do curso de Bacharelado em Ciência da Computação, Faculdade de Ciências, Unesp, câmpus de Bauru, como parte do Trabalho de Conclusão de Curso.

UNESP

2012

Gustavo André Arrabal de Souza

PROPOSTA DE ARQUITETURA E PROTOCOLO PARA A GERAÇÃO AUTOMÁTICA DE ASSINATURAS DE *MALWARES*

Monografia apresentada junto à disciplina Projeto e Implementação de Sistemas II, do curso de Bacharelado em Ciência da Computação, Faculdade de Ciências, Unesp, câmpus de Bauru, como parte do Trabalho de Conclusão de Curso.

BANCA EXAMINADORA

Prof. Dr. Marcos Antônio Cavenaghi
Professor Doutor
DCo - FC - UNESP - Bauru
Orientador

Prof. Dr. Renê Pegoraro
Professor Doutor
UNESP - Bauru

Profa. Dra. Simone das Graças Domingues Prado
Professora Doutora
UNESP - Bauru

Bauru, 29 de Outubro de 2012.

Dedicatória

Dedico este trabalho a meus pais, irmãos, namorada e amigos, que direta ou indiretamente contribuiram para a construção dessa monografia.

Agradecimentos

Agradeço primeiramente aos meus pais por terem me aconselhado e educado da melhor forma possível, ensinando-me sempre a seguir pelo caminho correto. Todas as palavras contidas em dicionários seriam insuficientes para expressar tamanha gratidão. Não posso deixar de agradecer, também a eles, pelo apoio fornecido para enfrentar essa jornada de 4 anos. Sou muito grato a tudo que fizeram para permitir, que hoje, eu seja quem sou.

Não poderia deixar de faltar nesses agradecimentos os nomes Guilherme e Gabriel que, mais do que irmãos, são meus melhores amigos, sempre dispostos a me ajudar. Espero que esses nunca esqueçam o código anticlonagem.

Seria impensável não citar minha melhor amiga, companheira e namorada, Carmen, que sempre esteve ao meu lado em Bauru, me apoiando e ficando aflita nas noites de estudo em que eu ficava sem dormir. A presença dela ao meu lado sempre foi essencial e fundamental para enxegar a vida por outro ângulo.

Também agradeço ao meu amigo Rodrigo Nascimento por sempre estar disposto a conversar e a me ajudar.

Não menos importante, gostaria de agradecer ao Saraiva (Henrique Ferraz) por ter me suportado nesses 4 anos e ter sido companheiro em muitas noites de estudo. Acredito que ambos amadureceram muito desde 2009. Nessa lista também incluo o Zuna (Daniel Vielmas), quem também nunca me negou ajuda e sempre foi muito solícito.

Sem ordem de importância, também adiciono nessa lista: Peixinho (Alan Peixinho); Stand (Daniel Felipe); Baiano (Felipe Avelar); X3 (Filipe Araújo); Hiro (Hiroaki); Ivan; Salomão (Lucas Salmen) e Marcel.

Os oito nomes acima, juntos com o Saraiva, Zuna e Carmen formam, o que considero de, minha família bauruense.

Não poderia deixar de agradecer ao meu professor Marcos Antônio Cavenaghi por sempre ter me orientado na construção desse trabalho. Sua ajuda e opiniões foram e sempre serão de extrema importância para mim. Também agradeço pela paciência e compreensão despendidas.

“Um homem que não se dedica à família nunca será um homem de verdade.”
Don Vito Corleone

Resumo

Dado o crescimento exponencial na propagação de vírus pela rede mundial de computadores (*Internet*) e o aumento de sua complexidade, faz-se necessária a adoção de sistemas mais complexos para a extração de assinaturas de *malwares* (assinatura de *malwares - malicious software;* é o nome dado a extração de informações únicas que levam a identificação do vírus, equivalente, aos humanos, a impressão digital). A arquitetura e o protocolo aqui propostos têm como objetivo tornar mais eficientes as assinaturas, através de técnicas que tornem suficiente uma única extração para comprometer todo um grupo de vírus. Essa eficiência se dá pela utilização de uma abordagem híbrida de extração de assinaturas, levando-se em consideração a análise do código e do comportamento do *sample*, assim também chamado um vírus.

Os principais alvos desse sistema proposto são *Polymorphics* e *Metamorphics Malwares*, dada a dificuldade em se criar assinaturas que identifiquem toda uma família proveniente desses vírus. Tal dificuldade é criada pelo uso de técnicas que possuem como principal objetivo comprometer análises realizadas por especialistas.

Os parâmetros escolhidos para realizar a análise comportamental são: Sistema de Arquivos; Registros do *Windows*; *Dump* da RAM e chamadas a API. Quanto à análise do código, o objetivo é realizar, no binário do vírus, divisões em blocos, onde é possível a extração de *hashes*. Essa técnica considera a instrução ali presente e sua vizinhança, sendo caracterizada como precisa.

Em suma, com essas informações pretende-se prever e traçar um perfil de ação do vírus e, posteriormente, criar uma assinatura baseada no grau de parentesco entre eles (*threshold*), cujo objetivo é o aumento da capacidade de detecção de vírus que não façam parte da mesma família.

Palavras-chave: Detecção, *Malware*s, assinatura, comportamento.

Abstract

Given the exponential growth in the spread of the virus worldwide web (Internet) and its increasing complexity, it is necessary to adopt more complex systems for the extraction of malware fingerprints (malware fingerprints - *malicious software;* is the name given to extracting unique information leading to identification of the virus, equivalent to humans, the fingerprint). The architecture and protocol proposed here aim to achieve more efficient fingerprints, using techniques that make a single fingerprint enough to compromise an entire group of viruses. This efficiency is given by the use of a hybrid approach of extracting fingerprints, taking into account the analysis of the code and the behavior of the sample, so called viruses.

The main targets of this proposed system are Polymorphics and Metamorphics Malwares, given the difficulty in creating fingerprints that identify an entire family from these viruses. This difficulty is created by the use of techniques that have as their main objective compromise analysis by experts.

The parameters chosen for the behavioral analysis are: File System; Records Windows; RAM Dump and API calls. As for the analysis of the code, the objective is to create, in binary virus, divisions in blocks, where it is possible to extract hashes. This technique considers the instruction there and its neighborhood, characterized as being accurate.

In short, with this information is intended to predict and draw a profile of action of the virus and then create a fingerprint based on the degree of kinship between them (threshold), whose goal is to increase the ability to detect viruses that do not make part of the same family.

Keywords: *Detection, Malwares, fingerprint, behavior.*

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

Diariamente, pessoas de todo o mundo fazem uso de computadores para diversas finalidades, seja para pesquisa, assistir vídeos, conversar com outras pessoas, entre outras coisas. Porém, muitas dessas pessoas não sabem que, diariamente, um único computador pode sofrer mais de 1900 ataques por minuto (GOSTEV, 2012). Esses ataques têm por objetivo infectar o computador da vítima com códigos maliciosos, afim de transformá-lo em uma máquina zumbi. É considerada uma máquina zumbi aquela que obedece a uma outra, principal (chamada *'master'*), sem o consentimento de seu dono. Essa máquina passará a fazer parte de uma rede de distribuição de *spams*, vírus, etc (KASPERSKY, 2012a).

O número de máquinas zumbis vem crescendo a cada dia (DAMBALLA, 2010), isso porque a disseminação de códigos maliciosos é uma prática apoiada e financiada por grupos *cyber* criminosos que se utilizam de estratégias do tipo *Pay-Per-Install*[1] (*PPI*) (STEVENS, 2010) para propagar vírus e, assim, criar suas *botnets*: a figura 1 mostra um exemplo do site que faz esse tipo de disseminação.

Botnet é o nome que se dá a rede de computadores, composta por máquinas zumbis e mestras, controlada por *cyber* criminosos para a disseminação de vírus e *spams*, sem a permissão de seu proprietário (KASPERSKY, 2012b). A figura 2 mostra uma rede formada por uma *botnet*.

[1] *PPI* é a técnica onde sites pertencentes a *cyber* criminosos disponibilizam *rootkits* (tipo especial de *malware - malicious software* ou programa malicioso) para *download* afim de que *sites* infectem seus visitantes. Com isso, a cada instalação desse *rootkit* com sucesso, o usuário receberá um valor em dólares do grupo *cyber* criminoso como forma de recompensa.

Figura 1: *Web Site Dogma Million* - Site que faz pagamentos a quem contribui disseminando os seus *rootkits* (MATROSOV; RODIONOV, 2012).

Se já não fosse suficiente, esse método para a infecção de máquinas, hoje, qualquer pessoa é capaz de invadir e infectar uma máquina vulnerável[2], bastando apenas um computador com acesso a Internet. Isso acontece porque a quantidade de informações sobre como efetuar uma invasão - seja em forma de tutoriais, vídeos, algoritmos, etc, estão facilmente disponíveis na Internet, a fácil alcance, possibilitando o ato.

Atualmente uma das maiores preocupações de empresas em relação aos seus *data centers* é com a proteção de dados sigilosos neles contidos. Por conta disso, a necessidade em investimentos na área de segurança da informação cresce a cada ano (TIINSIDE, 2011). No entanto, como nenhum sistema é totalmente seguro, pode acontecer de pessoas mal intencionadas e com alto conhecimento agregado invadirem o sistema e tomarem posse de informações confidenciais, seja para posterior venda em mercados negros ou para cumprir serviços contratados por empresas concorrentes.

[2]Entende-se por "máquina vulnerável" aquela que não satisfaz os pré-requisitos mínimos de segurança; ou seja, que possui *softwares* e antivírus desatualizados e que falha no controle de suas portas e fluxos de dados.

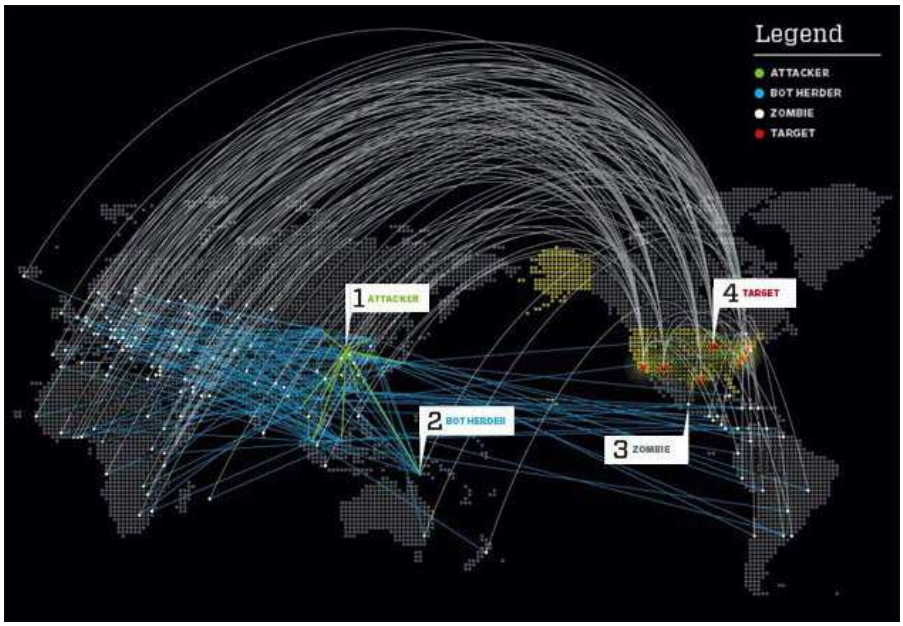


Figura 2: Exemplo de uma rede *Botnet (SOCIALENGINEER, 2012).*

Vale salientar que, para as empresas, a velocidade de análise a incidentes é muito importante, pois encontrar o problema e solucioná-lo com maior agilidade significa menores danos financeiros para uma organização e menor tempo de exposição da falha, podendo evitar maiores danos (seja ele por roubo de informações sigilosas ou pelo uso indevido do equipamento para outros fins, como a criação de um servidor de email para *spam*, por exemplo).

1.1 Problema

Devido a enorme quantidade de ameaças *online*[3] , aliada ao crescente número de ataques efetuados em máquinas conectadas a Internet (YURY, 2012). Constata-se, portanto, a necessidade de se ter conhecimento imediato da invasão e, em caso positivo, quais dados ficaram expostos (a figura 4 ilustra, quais as principais aplicações exploradas pelos criadores de *malwares*), tendo em vista que informações sempre estão agregadas a valores financeiros.

Para combater tamanho problema é comum que algumas empresas da área de tecnologia e outras focadas em segurança da informação se unirem em busca de uma solução comum, como ocorreu para a neutralização da *botnet Kelihos* (ou também *Waledac 2.0*) (BOSCOVICH, 2011). Essa *botnet* tinha a capacidade de enviar 3.8 bilhões de *emails* por dia e tinha à sua disposição aproximadamente 41 mil computadores zumbis ao redor do mundo.

Existem muitas outras *botnets*, ainda maiores, espalhadas pela Terra, algumas com capacidade de enviar mais de 60 bilhões de emails por dia e que controlam mais de 300 mil computadores (STEWART, 2008).

[3] *Kaspersky Lab* detectou e neutralizou mais de 1 bilhão de ameaças somente no segundo trimestre de 2012 (NAMESTNIKOV, 2012). A figura 3 mostra a distribuição dessas ameças pelo mundo.

Todas essas máquinas concentradas sob o comando de uma única organização criminosa pode gerar danos, por meio de ataques distribuidos de negação de serviço (*DDoS*) muito eficazes, possibilitando que o tráfego de Internet alcance a picos de 100Gbps - superior ao tráfego de muitos paises (CONSTANTIN, 2011).

Esses ataques geram prejuízos às organizações e aos seus clientes, sendo os últimos prejudicados por não terem à sua disposição o serviço de uma empresa, seja ela de vendas *online ou financeira,* ou devido às empresas não conseguirem manter os seus serviços disponíveis.

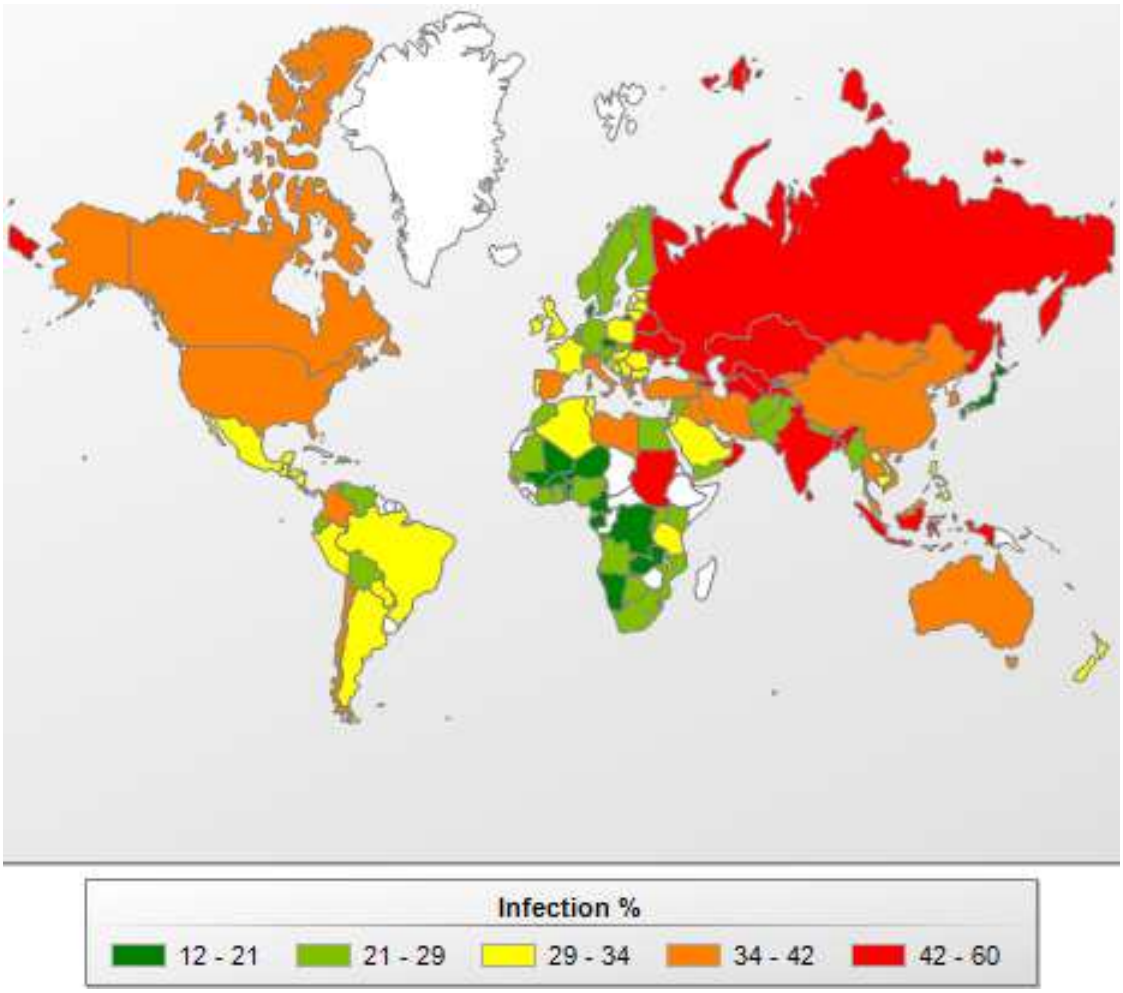


Figura 3: Ameaças Online: análise referente ao segundo trimestre de 2012 (NAMESTNIKOV, 2012).

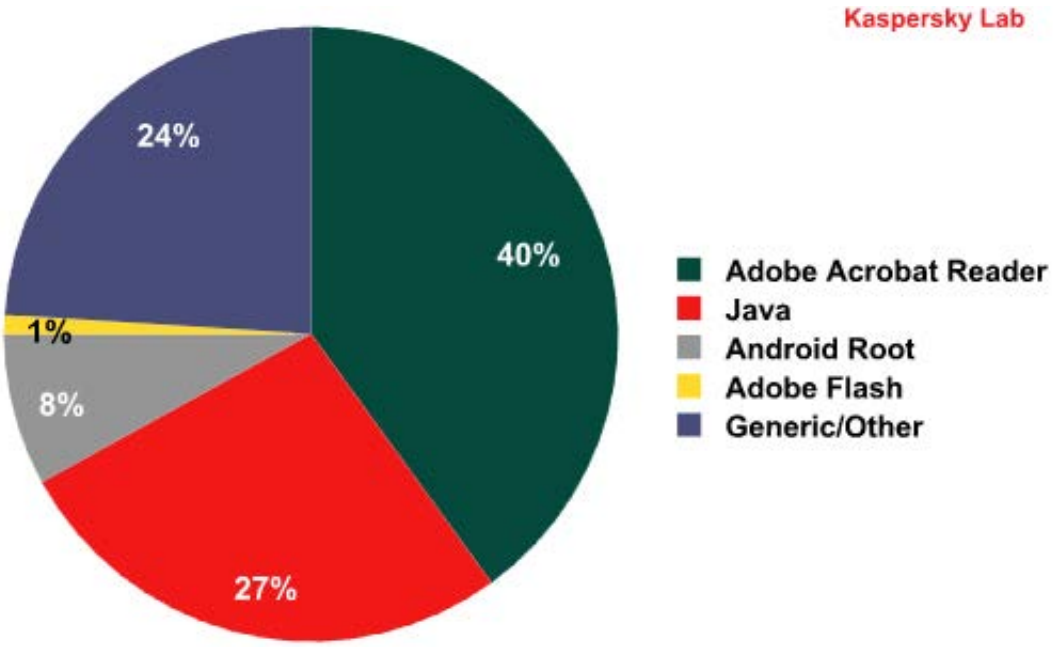


Figura 4: Principais aplicações que possuem falhas exploradas por criadores de *malwares* (NAMESTNIKOV, 2012).

Muitos desses *malwares* criados são de difícil detecção por se utilizarem de técnicas que, muitas vezes, não permitem seu estudo e que possuem a habilidade de auto-mutação, por meio do qual o *malware* pode se replicar de maneira distinta à anterior. Essas mutações dificultam a criação e implementação de regras em *IPSs* (*Intrusion Prenvent System* - Sistema de Detecção de Intrução) e em sistemas de anti-vírus.

IPSs são sistemas que possuem implementadas regras de detecção de *malwares* e que quando instalados em um ambiente de rede de computadores, monitoram todos os pacotes que ali trafegam afim de definir um padrão de normalidade (pacote normal ou malicioso) (BRANCO, 2007). Caso o pacote malicioso seja detectado o *IPS* deverá realizar seu bloqueio.

Já sistemas de anti-vírus que se baseiam somente em técnicas de *string matching* em assinaturas [4] de *malware* são incapazes de detectar programas maliciosos após sua mutação, por motivos que serão explicados em capítulo próprio.

1.2 Objetivos

Nessa seção será abordado o objetivo a ser conquistado por esse trabalho bem como quais passos culminaram essa conquista.

[4] O termo técnico "assinatura de vírus" e "*string matching*" serão explicados de forma detalhada no capítulo 3.

1.2.1 Objetivo Geral

Essa pesquisa tem como objetivo principal propor uma arquitetura física e um protocolo lógico para o auxílio na identificação de intrusões em sistemas computacionais. Tal protocolo será capaz de gerar assinaturas, de maneira automática, capazes de identificar *malware* provenientes de sistemas de mutação ou não.

Tais assinaturas são úteis em aplicações de regras em *IPSs*, bem como aplicações em banco de dados de sistemas de anti-vírus, em razão de serem geradas de forma híbrida, sendo capazes de usar informações tanto do código, quanto do comportamento do *malware*.

1.3 Metodologia

Para se atingir o objetivo final de criação de uma arquitetura e protocolo lógico que gere assinaturas de *malwares* automaticamente, a metodologia a ser seguida foi:

1. Estudar os sistemas de arquivos atuais (*NTFS* e *FAT*)
2. Compreender como os vírus poderão se dispor no sistema;
3. Estudar os registros do sistema operacional *Windows:*
   (a) como funcionam;
   (b) sua disposição em disco;
   (c) possíveis chaves utilizadas por *malwares* para permanecerem no sistema.
4. Estudar como os processos são dispostos, pelo sistema operacional, na memória RAM[5] e como extrair informações dela;
5. Estudar como funcionam as funções da *API* do *Windows* ;
6. Extrair as funções invocadas pelos *malwares*;
7. Estudar os tipos de vírus existentes, suas estruturas e como realizam suas replicações (se há ou não mutações);
8. Estudar as técnicas utilizadas pelos criadores de vírus para dificultar a análise do *malware* por especialistas;
9. Estudar como as empresas de anti-vírus trabalham afim de combater tais pragas.

[5]Random Access Memory

1.4 Organização do trabalho

Este trabalho está organizado da seguinte maneira: O próximo capítulo descreverá sobre os sistemas de arquivos *FAT* e *NTFS*, no capítulo 3 os Registros do *Windows* são abordados, perpassando a discussão de suas chaves específicas; no capítulo 4 é abordado como o sistema operacional organiza informações na memória RAM e algumas técnicas para realizar essa extração. Dados que trafegam na rede e como capturar pacotes são temas abordados no capítlo 5. Já no capítulo 6 é estudado como funciona a *API* do *Windows*. Finalmente no capítulo 7 é proposta a arquitetura e o protocolo lógico para a geração de assinaturas de *malware*. As considerações finais são feitas no capítulo 8.

Capítulo 2

Sistema Operacional *Windows*

2.1 Introdução

Este capítulo apresentará como o sistema operacional *Windows* gerencia mecanismos como: Sistema de Arquivos, Registros do Sistema, Alocação de processos na memória RAM e Utilização de recursos através de funções pertencentes a API.

Para uma análise de *malware* completa todas esses mecanismos devem ser observados e cada seção explica qual a importância das informações disponíveis para a criação de assinaturas de detecção de vírus.

2.2 Sistema de Arquivos

Sistema de arquivos é o modo pelo qual o Sistema Operacional (S.O.) lida com os arquivos. Em ambientes *Windows,* os sistemas de arquivos suportados são: *File Allocation Table (FAT)* e *New Technology File System (NTFS*). Ambos são explicados nas seções 2.2.1 e 2.2.2, respectivamente.

É fundamental frisar a importância do Sistema de Arquivos para uma análise de *malwares*. Analisando-se este sistema pode-se observar, em tempo real, quais arquivos e/ou diretórios estão sendo criados, modificados e/ou excluídos. Com essa informação obtem-se uma visão mais detalhada do perfil do *sample*[1] em questão, ou seja, se ele modifica/cria/exclui arquivos no sistema e quais arquivos são esses. De posse desse dado pode-se ir até tais arquivos e analisa-los de maneira mais minuciosa e por fim traçar o comportamento do *malware* em relação ao que ele influência no Sistema de Arquivos.

2.2.1 Sistema FAT

O sistema de arquivos *FAT* pode ser visto como o mais simples entre os demais. Seu funcionamento é baseado em sua versão*: FAT12, FAT16* e *FAT32* (CARVEY; ALTHEIDE, 2011):

[1]Define-se *sample* como sendo qualquer amostra de um dado *malware*.

Basicamente o funcionamento de sistemas *FAT* ocorre da seguinte maneira:

1. Existe uma pasta raíz (C:\, por exemplo) em um local pré-especificado que o S.O. saberá onde encontrá-lo, essa pasta terá a lista completa de seus arquivos e subdiretórios.
2. A *FAT* mapeia todos os clusters do volume e informa ao S.O. como o *cluster* é utilizado (no disco, o sistema *FAT* inicia no setor de *boot*), ou seja, se a entrada for um número igual a 0 pode-se concluir que o *cluster* nunca foi alocado ou que o arquivo que existia, em tal *cluster*, foi excluído. Por outro lado se o número for maior que 0 indica que o *cluster* está ocupado. Todas essas informações estão presentes na *FAT*;
3. Os arquivos são alocados nos *clusters*;
4. Uma entrada de 32 *bytes* é criada no diretório raiz para armazenar o endereço do *cluster* que tal arquivo se inicia;
5. Se o arquivo ocupa mais de um *cluster* então o primeiro terá como informação o número do *cluster* seguinte, e essa forma de funcionamento se repete até que se encontre um *end-of-file* (*EOF*).

Para exemplificar todo esse processo, suponhamos que iremos ler um arquivo “A”:

1. O diretório raiz indica que esse arquivo inicia no *cluster* 153;
2. O S.O. vai até o campo referente ao *cluster* 153 na *FAT*;
3. Lá encontra o número 154, ou seja, o arquivo “A” inicia no *cluster* 153 e continua no 154;
4. Ao se dirigir ao campo 154 da *FAT*, o S.O. se depara com o número 162, ou seja, o arquivo continua no *cluster* 162;
5. Ao chegar no campo 162 da *FAT* o S.O. se depara com um <EOF>, ou seja, o arquivo terminou no *cluster* 162. (CASEY, 2011)

A figura 5 representa a *FAT*, atenção para o que está em vermelho, temos um arquivo iniciando no *cluster* 632, com seu conteúdo apontando o cluster 633 como o próximo que deve ser lido (continuação do arquivo) e assim por diante até que o mesmo finaliza (<EOF>) no cluster 643.

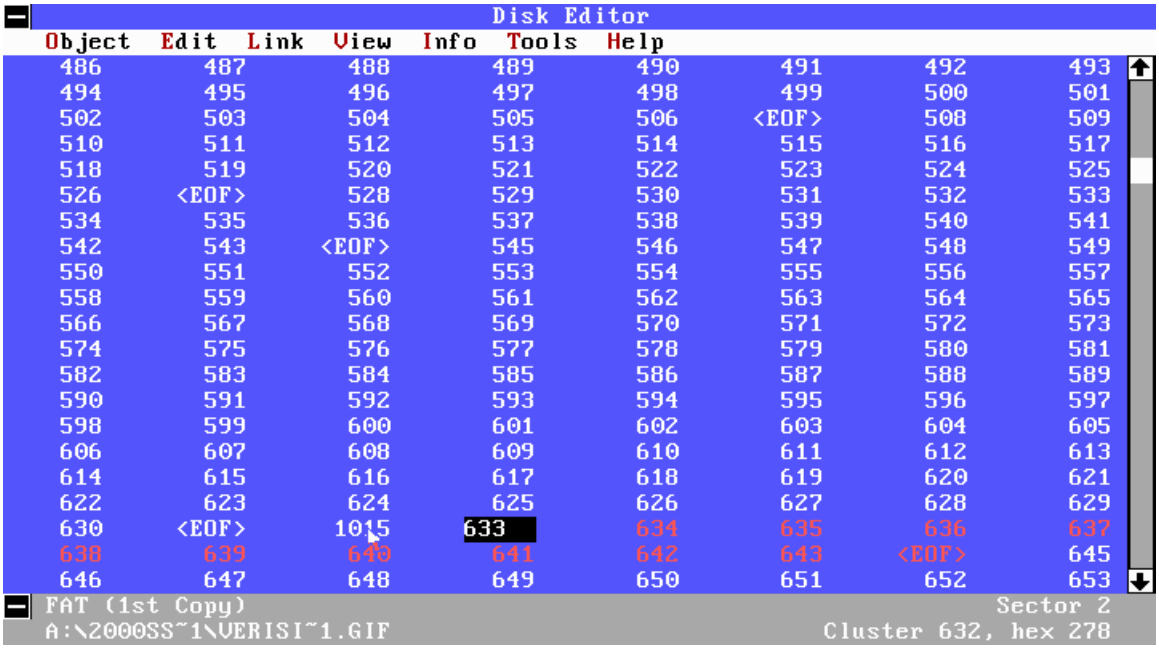


Figura 5: *Clusters* listados na *FAT* (INFORMIT, 2002).

A figura 6 mostra o que acontece na *FAT* quando o mesmo arquivo (em vermelho) da figura 5 é excluído. Vale ressaltar que quando um arquivo é exluído e sua entrada na *FAT* é sinalizada como "*cluster* livre", ou seja, é colocada em 0 (zero) não necessariamente o arquivo no HD foi perdido. Até que um novo arquivo sobreponha o arquivo excluído no HD o mesmo continuará presente mesmo sem ter uma entrada na *FAT* e isso significa que arquivos excluídos por vírus podem ser recuperados para análise.

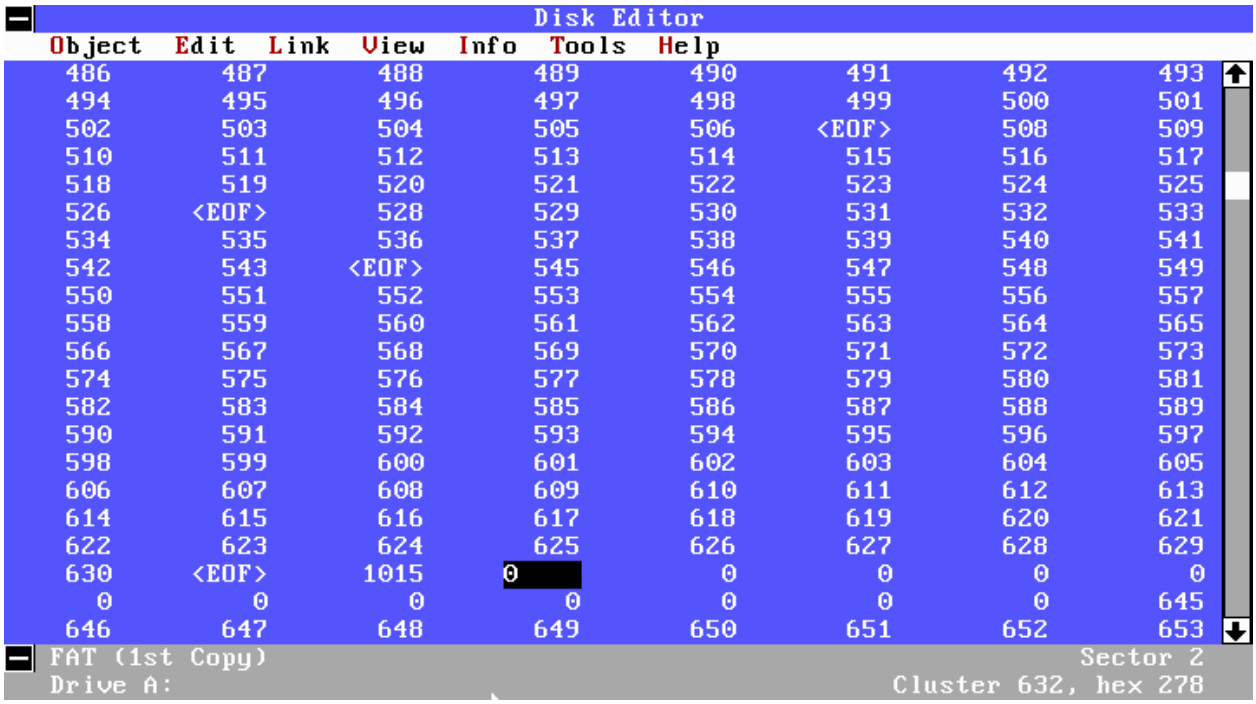


Figura 6: Arquivo excluido da *FAT* (INFORMIT, 2002).

Esse sistema foi atualmente substituído pelo *NTFS*, porém ele não está em total desuso. Sua substituição ocorreu devido as limitações e outros problemas existentes.

Entre as desvantagens desse sistema de arquivos estão: limitação que este implica ao tamanho do HD (2TB ou 16TB para setores de 4KB), problema com fragmentação de disco (o processo de exclusão de um arquivo e a criação de um novo cria espaços no disco que se tornarão inutilizáveis, isso torna o processo de leitura e escrita cada vez mais lento. Para se resolver esse problema foi implementado o processo de desfragmentação, porém esse é um processo demorado) e segurança (um sistema com muitos usuários não impede que um usuário "A" deixe de ler e/ou modificar arquivos criados por outros usuário) (CARVEY; ALTHEIDE, 2011).

2.2.2 Sistema NTFS

Entender o *NTFS* após se ter compreendido o sistema *FAT* é mais elementar. Nesse sistema diversos arquivos fazem o gerenciamento do sistema porém, um arquivo exerce essa função de modo decisivo: o *Master File Table (MFT)*, que está armazenado no setor de *boot*. Cada entrada nessa tabela (*MFT*) representa um arquivo, pasta ou diretório (esse último, chamado de "*index entries*") e todas essas entradas são armazenadas em uma árvore B, dentro da *MFT*, para catalisar as buscas e aprimorar a organização (CASEY, 2011). Uma entrada na *MFT* é composta por diversos atributos (obedecendo a estrutura da figura 7) que armazenam informações diversas sobre o arquivo. Entre os atributos existentes podemos destacar três (NTFS, 2012):

- *$Standard_Information*: Possue armazenada diversas informações úteis para análise, porém,

um dado importante são os *timestamps*[2] do arquivo;

- *$File_Name*: Mantém os mesmos *timestamps* do atributo anterior que, contudo só serão atualizados na ocasião de mudança do nome do arquivo;
- *$Data*: Pode conter o conteúdo do arquivo ou a informação sobre onde encontrar esse conteúdo em disco, o que demonstra a sua importância.

O cabeçalho da entrada na *MFT* (42 *bytes* iniciais) contém alguns componentes (por exemplo: *Signature, Sequence Value, Link Count, Flags* e etc) e dois merecem uma atenção especial:

- *Signature*: É o primeiro componente da entrada, este pode ter em seu conteúdo a *string* "*FILE*" ou "*BAAD*" (se houver algum erro na entrada, a *string* será "*BAAD*");
- *Flags*:
  - 0x00: A entrada não está em uso (o arquivo pode ter sido apagado, por exemplo. Arquivos apagados têm sua flag modificada para 0x00 e não têm sua entrada apagada da *MFT*);
  - 0x01: Entrada em uso;
  - 0x02: A entrada é para um diretório ou pasta;
  - 0x03: Entrada para diretório ou pasta em uso.

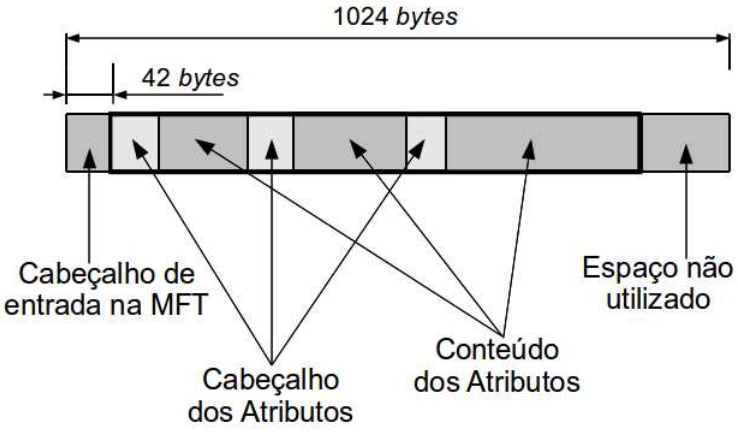


Figura 7: Estrutura da entrada na MFT.

Portanto qualquer arquivo criado (nova entrada é criada na *MFT*), alterado (a data de modificação no atributo *$Standard_Information* é modificada) ou deletado (a componente "*Flag*" do cabeçalho é alterada para 0x00) passará por uma análise minuciosa, em que todas as modificações são armazenadas em relatórios que, posteriormente, servirão como parâmetro na criação da assinatura do *malware*.

[2] *Timestamp* são todas as informações relacionadas a datas e horários do arquivo, seja de criação, alteração, exclusão, mudança de *path*, leitura e etc.

Por fim temos uma tabela comparativa entre o Sistema *FAT* e *NTFS* (tabela 1), onde podemos ver de forma sintética as especificidades inerentes a tais sistemas (*FAT* x *NTFS*).

Tabela 1: *NTFS* x *FAT* (NTFS, 2011).

Critério	*NTFS*	*FAT32*
Sistemas Operacionais	*Windows 2000* *Windows XP* *Windows 2003 Server* *Windows 2008* *Windows Vista* *Windows 7*	*DOS v7 and higher* *Windows 98* *Windows ME* *Windows 2000* *Windows XP* *Windows 2003 Server* *Windows Vista* *Windows 7*
Tamanho máximo do volume	2^{64} *clusters* menos 1 *cluster*	32GB para todos os S.O. e 2TB para alguns.
Quantidade máxima de *clusters*	2^{64} *clusters* menos 1 *cluster*	65520
Tamanho máximo do arquivo	2^{64} *bytes* (16 *ExaBytes*) menos 1KB	4GB menos 2 *Bytes*
System Records Mirror	Arquivo *MFT* espelhado	Segunda cópia da *FAT*
Sistema de permissão de arquivos	SIM	NÃO

2.3 Registros do *Windows*

Uma análise de *malware* em ambientes *Windows* não pode deixar de lado os registros de tal sistema operacional. Registros do *Windows* é um banco de dados hierárquico central (WINDOWS. . . , 2008), isso significa que absolutamente tudo que ocorre no sistema, com ou sem permissão ou ciência do usuário, será armazenado nesse banco de dados. Nele estão contidas todas as informações necessárias para configurar o sistema para um ou mais usuários, aplicações e dispositivos de hardware. Esse também armazena históricos de tudo que um dia foi instalado e desinstalado no computador, sendo sua funcionalidade semelhante ao de um arquivo de *log*. Consequentemente, as informações mantidas nos registros do *Windows* adquirem importância para uma análise completa do comportamento de um vírus.

Atualmente os registros do *Windows* possuem cinco tipos de chaves raízes (como visto na tabela 2), que juntas formam o registro do sistema.

Tabela 2: Tipos de chaves raízes do *Windows*.

Nome	Abreviação
HKEY_CLASSES_ROOT	HKCR
HKEY_CURRENT_USER	HKCU
HKEY_LOCAL_MACHINE	HKLM
HKEY_USERS	HKU
HKEY_CURRENT_CONFIG	HKCC

Cada uma dessas chaves raízes possui a sua funcionalidade específica:

- *HKU*: armazenar informações específicas de todos os usuários do sistema;
- *HKCU*: armazenar informações de configuração do usuário que está utilizando o computador no momento. Por exemplo: pastas do usuário, cores das janelas, imagens de plano de fundo, configurações do painel de controle e etc. Atualmente essa chave nada mais é do que um *link* para a chave *HKU*, ou, pode-se dizer, que essa chave é uma subchave da *HKU*;
- *HKLM*: armazenar as configurações que serão aplicadas a todos os usuários, como por exemplo: informações do *hardware*, informações de segurança, *drivers* instalados, etc;
- *HKCR*: Atualmente, essa chave é uma subchave da *HKEY_LOCAL_MACHINE\Software*. Toda informação armazenada aqui garante que, no momento em que o usuário abrir um arquivo o programa correto será executado;
- *HKCC*: Também é um *link* para a chave *HKLM*, a qual mostra as configurações de *hardware* (HONEYCUTT, 2005).

Além das chaves pode-se notar na figura 8 que a representação gráfica dos registros estão estruturados em sub-chaves e *values*, sendo que este último armazena as informações necessárias (fazendo-se uma analogia com a estrutura do *Windows Explorer* pode-se dizer que as chaves seriam as pastas, sub-chaves as sub-pastas e os *values* os arquivos que armazenam as informações necessárias).

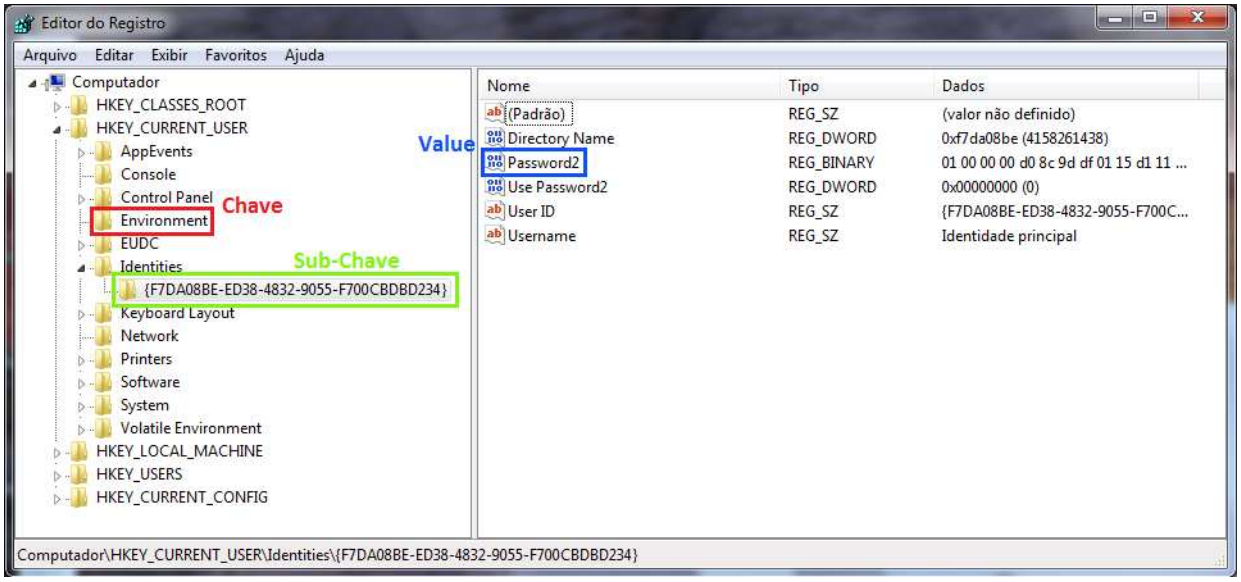


Figura 8: Estrutura dos Registros.

Não são todas as chaves raízes que se encontram armazenadas em disco, as chaves que são somente *links* para outra(s) (*HKCU, HKCR, HKCC*) não tem o seu conteúdo armazenado (por serem somente ponteiros) exclusivamente.

O armazenamento em disco do Registro do *Windows* é feito em arquivos denominados "arquivos *hives*". *Hives* são arquivos binários que armazenam todas as informações do sistema sendo

que cada parte de um registro pode estar contido em diferentes arquivos *hives* (chaves e sub-chaves que estão com seus nomes inteiro em letras maiúsculas possuem seus *hives* exclusivos). Muitos desses arquivos se encontram em *system32/config/* (CARVEY, 2011b). A figura 9 faz uma ligação de algumas chaves com seus *hives* exclusivos.

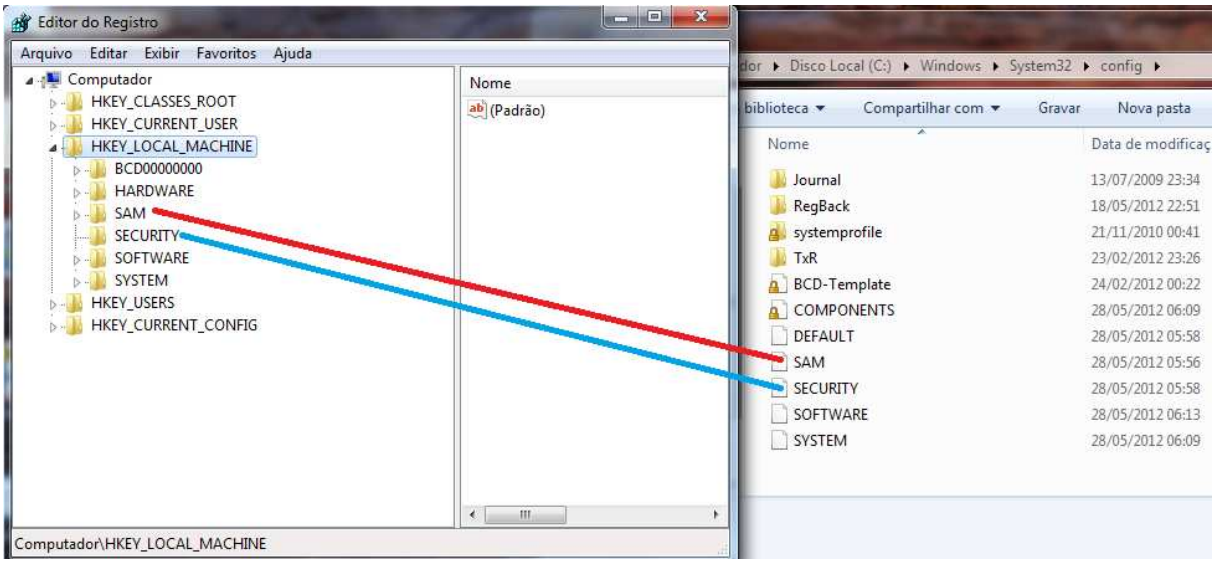


Figura 9: Chaves e seus respectivos arquivos *hives*.

Dessa forma afirma-se que o coração de todas as configurações do sistema se encontram em arquivos *hives* e de maneira visual estruturado em forma de chaves e sub-chaves.

Dada a sua importância muitos vírus fazem dos registros um meio para se tornarem persistentes, ou seja, qualquer ação (desligar, reiniciar, desinstalar/instalar programas, limpeza de registros, etc) feita no computador não fará com que o *malware* deixe de ser executado (CASEY, 2011). Para tal o vírus pode se utilizar de chaves como a "*(...)\Software\Microsoft\Windows\CurrentVersion\Run*" para se manter ativo no sistema.

A chave *Run* tem a função de executar todas aplicações listadas em seus *values* e isso será feito sem a ciência do usuário. Tal atividade poderá ocorrer no monento de *boot, logon, logoff* ou qualquer outra ação pré-definida. Deve-se atentar para dois fatores, o primeiro é que independente de quantas vezes o *boot, logon* ou qualquer outra atividade ocorrer o *malware* será reexecutado (existe a chave *RunOnce* e essa tem a funcionalidade idêntica ao da chave *Run* com a única diferença que todos os programas ali listados somente serão executados no proximo *boot* e logo depois terão seus *values* deletados), o segundo fator é que existe a possibilidade de configurar tais *values* para que o mesmo seja executado até em *boots* realizados em *Safe Mode* (CARVEY, 2011b).

Portanto pode-se concluir que ao se analisar um *sample* deve-se atentar para as mudanças que o mesmo faz nos registros do sistema pois tais alterações contribuirá posteriormente na criação de uma assinatura.

2.4 *Dump* da RAM

Adiantado-se ao capítulo que discorre sobre vírus, é importante esclarecer que alguns *malwares*, ao se intalarem nas máquinas alvo, alocam-se na memória do computador e se auto-destroem do *HD*, quer dizer, qualquer busca feita exclusivamente em disco se torna em vão pois o mesmo esta presente somente na *RAM*. Assim, visualiza-se de antemão a importância de uma análise da memória *RAM* de modo minucioso, afim de se encontrar um vírus residente e também para se descobrir o que estes e outros vírus estão manipulando em tal dispositivo.

O termo *dump* da *RAM* é utilizado para técnica de extração do conteúdo da memória. Essa extração gera como saída um arquivo binário com todas as informações presentes no componente no momento da captura. Entender como o *Windows* organiza os processos em execução na memória é essencial para se conseguir uma análise efetiva do binário.

Após um arquivo ser executado no sistema o *Windows* cria um bloco chamado *EProcess* (esse é explicado mais a frente), referente ao processo. Esse bloco é disposto na memória de maneira circular, ou seja, dentro desse bloco há diversos atributos (alguns desses vistos mais adiante) e dois deles são o *FLink* (*Forward Link*) e o *BLink (Back Link):*

- *FLink*: ponteiro para o *EProcess* do processo seguinte disposto na RAM;
- *BLink*: ponteiro para o bloco do processo anterior;

Compondo então uma lista circular (CARVEY, 2011a). A figura 10 ilustra de maneira clara como isso ocorre, supondo que “A”, “B” e “C” sejam os blocos *EProcess* dos processos “A”, “B” e “C” respectivamente. Na figura 10, além dos ponteiros *FLink* e *BLink*, visualiza-se mais algumas informações armazenadas no *EProcess*.

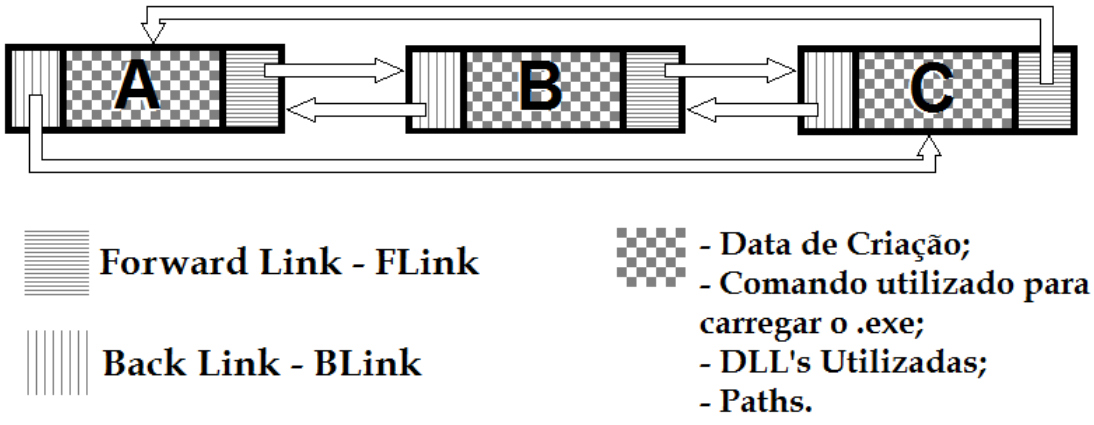


Figura 10: Disposição, realizada pelo *Windows* , dos *EProcess* na memória.

Compreender como essa disposição ocorre é de suma importância para se extrair os dados do *dump* da RAM de modo correto.

2.4.1 Abordagens de Extração

Para a estração de informações do binário gerado após o *dump* da RAM é utilizada ou a técnica "Varredura da Lista" ou a "Força Bruta". Portanto para se obter êxito nesse processo é importante conhecer como funciona cada uma dessas abordagens.

2.4.1.1 Varredura da Lista

Uma das alternativas para extração das informações existentes no *dump* é procurar por um *EProcess* e, ao encontrá-lo percorrer toda a lista através dos ponteiros até que se retorne ao ponto de partida, garantindo que se tenham extraído todos os blocos presentes na lista circular. Esse método de busca é conhecido como "varredura da lista" e, por mais eficaz que pareça, leva consigo um grande problema: listar somente os *EProcess* presentes na lista.

Para conseguir evidenciar o problema é preciso saber que os executáveis em geral, quando construídos, estão configurados por padrão a entrarem nessa lista circular, porém alguns *malwares* são projetados para não fazerem parte de tal lista e isso os fazem passar despercebidos em análises que se utilizam da abordagem de varredura da lista. A figura 11 exemplifica esse caso, nela observa-se o *EProcess* "M" fora da lista circular.

2.4.1.2 Força Bruta

Para corrigir tal falha foi desenvolvida uma abordagem para vasculhar todo o arquivo binário resultante do *dump* e assim extrair efetivamente todos os *EProcess*, independente se estiver ou não na lista circular. Essa técnica, por ser mais precisa, consome mais tempo e mais recursos do sistema. Tal abordagem é conhecida como "Força Bruta" isso pelo motivo de ela vasculhar todo o arquivo, byte-a-byte, atrás de resultados (CARVEY, 2011a).

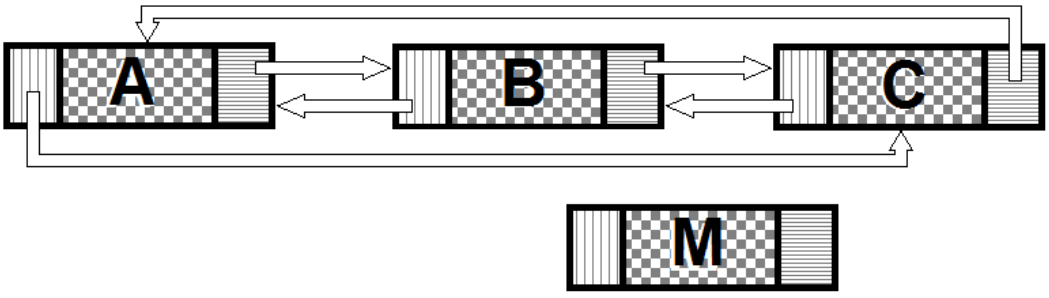


Figura 11: Bloco *EProcess* pertencente a um *malware* fora da lista circular.

2.4.2 Criação do processo

Para entender melhor como funciona a criação de um "bloco" *EProcess*, são abordadas as etapas ocorridas desde o momento em que realiza-se um duplo clique em um arquivo executável .

Do momento de execução do arquivo até que fique disponível para o usuário, o *Windows* terá executado 6 etapas (SOLOMON; RUSSINOVICH, 2000), conforme a figura 12. Todas serão explicadas sem a intenção de aprofundamento.

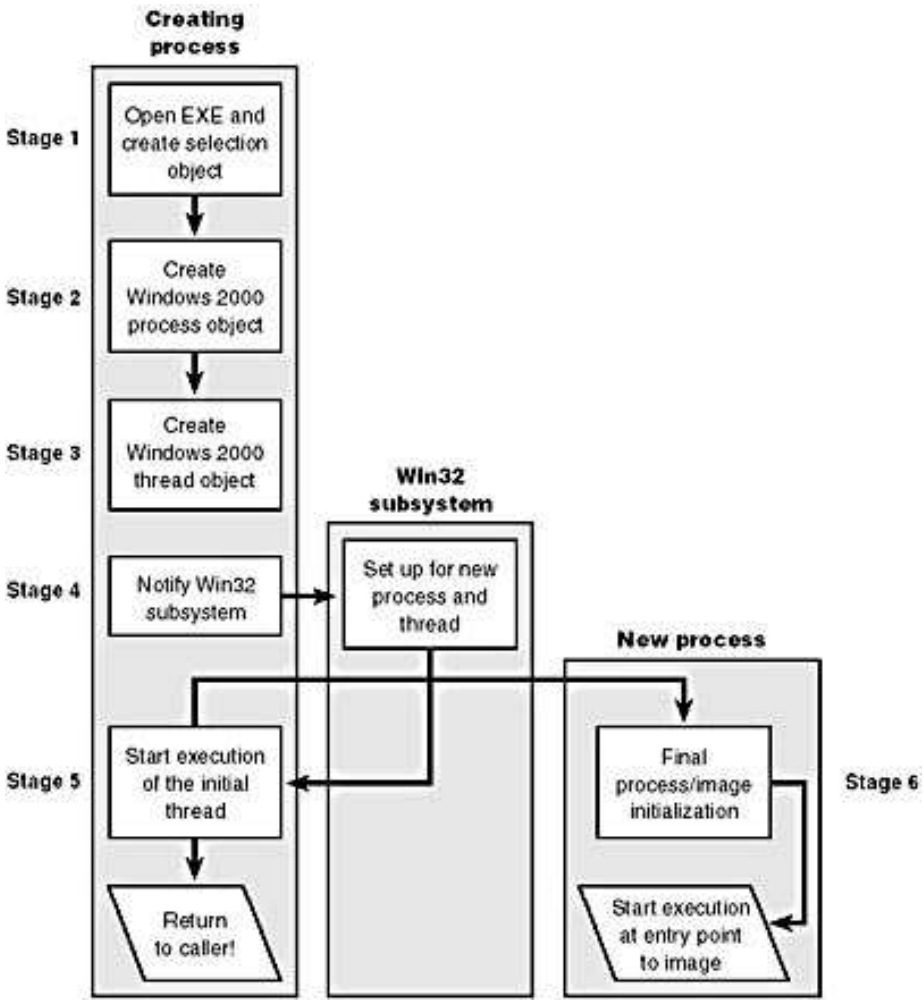


Figura 12: Estágios seguidos pelo *Windows* para a criação de um processo (SOLOMON; RUSSINOVICH, 2000).

- Etapa 1: Momento em que o arquivo é executado. Nesta etapa, o *Windows* tenta identificar a imagem apropriada do arquivo executado, conforme pode-se observar na figura 13.

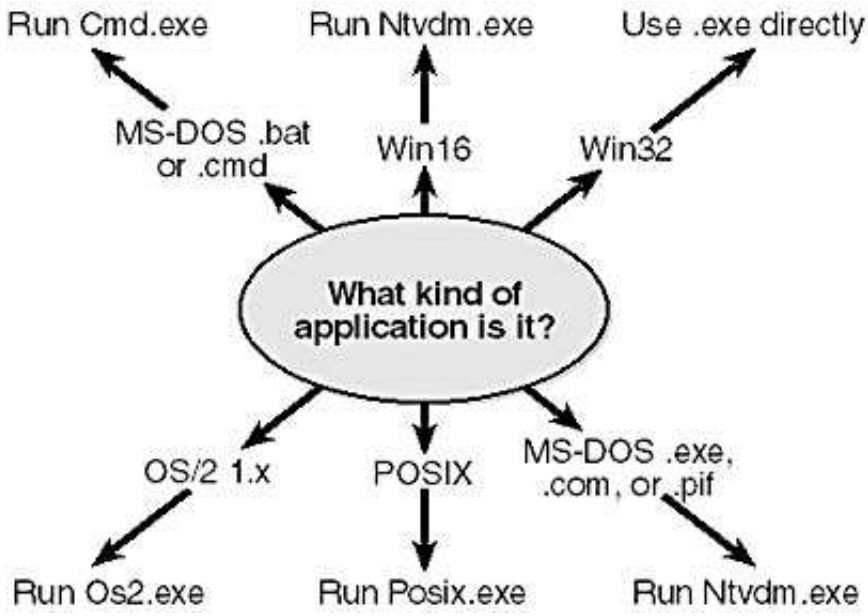


Figura 13: Escolhendo a imagem apropriada para a aplicação (SOLOMON; RUSSINOVICH, 2000).

- Etapa 2: Nessa etapa ocorre:

  1. Configuração do *Executive* Process (*EProcess*)*;*
  2. Alocação de espaço na memória;
  3. Criação do *Kernel Process* (*KProcess*);
  4. Conclusão do processo de alocação de memória;
  5. Configuração do *Process Environment Block* (*PEB*)*;*
  6. Conclusão da configuração do *EProcess.*

- Etapa 3: A *thread* inicial é criada;
- Etapa 4: O subsistema do *Windows* é notificado da criação de um novo processo e uma nova *thread*;
- Etapa 5: Inicia-se a execução da *thread*: o processo é estabelecido e os recursos necessários para o uso estão alocados;
- Etapa 6: O processo de alocação de espaço na memória está completo, as *DLLs* necessárias estão carregadas. Por fim, inicia-se a execução do programa.

Nesse momento, se o *dump* da *RAM* for efetuado será obtido o conteúdo a ser analisado.

Dentre as essas informações citadas nas etapas acima, o bloco *EProcess* merece especial atenção. A partir de agora, será estudada a estrutura desse bloco e seu funcionamento.

2.4.2.1 *Executive Process - EProcess*

A figura 14 mostra uma parte de como está o *EProcess* na *RAM*. Tendo em vista os dados que podem ser extraídos desse bloco afim de colaborar com a criação de uma assinatura para um dado *malware,* pode-se analisar alguns parâmetros:

- A data de criação do processo pode ser vista na linha 101 da figura 14 (CARVEY, 2011a);
- Ainda na figura 14 pode-se extrair da linha 199 um ponteiro que indica a posição da estrutura do *PEB* (figura 15), no qual se pode obter informações importantes como :
  - Na linha 8 da figura 15 pode-se extrair os nomes de todas as *DLLs* que foram carregadas e suas respectivas *paths*(MICROSOFT, 2012c);
  - A *path* do executável bem como o comando inserido para executar tal arquivo está presente na linha 9 da figura 15 também (MICROSOFT, 2012e);
- Os ponteiros indicando para o *EProcess* seguinte e o anterior da lista circular, presente na memória, podem ser visualizados nas linhas 294 e 295 respectivamentes da figura 14 (CARVEY, 2011a).

```
kd> dt -b -v _EPROCESS
struct _EPROCESS, 125 elements, 0x270 bytes
   +0x000 Pcb              : struct _KPROCESS, 35 elements, 0x80 bytes
      +0x000 Header           : struct _DISPATCHER_HEADER, 13 elements, 0x10 bytes
         +0x000 Type             : UChar
         +0x001 Abandoned        : UChar
         +0x001 Absolute         : UChar
         +0x001 NpxIrql          : UChar
         +0x001 Signalling       : UChar
         +0x002 Size             : UChar
         +0x002 Hand             : UChar
         +0x003 Inserted         : UChar
         +0x003 DebugActive      : UChar
         +0x003 DpcActive        : UChar
         +0x000 Lock             : Int4B
         +0x004 SignalState      : Int4B
         +0x008 WaitListHead     : struct _LIST_ENTRY, 2 elements, 0x8 bytes
            +0x000 Flink            : Ptr32 to
            +0x004 Blink            : Ptr32 to
      +0x010 ProfileListHead  : struct _LIST_ENTRY, 2 elements, 0x8 bytes
         +0x000 Flink            : Ptr32 to
         +0x004 Blink            : Ptr32 to
      +0x018 DirectoryTableBase : Uint4B
      +0x01c Unused0          : Uint4B
(...)
      +0x000 Shared           : Bitfield Pos 4, 28 Bits
      +0x000 Value            : Uint4B
      +0x000 Ptr              : Ptr32 to
   +0x088 CreateTime       : union _LARGE_INTEGER, 4 elements, 0x8 bytes
      +0x000 LowPart          : Uint4B
      +0x004 HighPart         : Int4B
      +0x000 u                : struct , 2 elements, 0x8 bytes
         +0x000 LowPart          : Uint4B
         +0x004 HighPart         : Int4B
      +0x000 QuadPart         : Int8B
(...)
   +0x180 DefaultHardErrorProcessing : Uint4B
   +0x184 LastThreadExitStatus : Int4B
   +0x188 Peb              : Ptr32 to
   +0x18c PrefetchTrace    : struct _EX_FAST_REF, 3 elements, 0x4 bytes
      +0x000 Object           : Ptr32 to
      +0x000 RefCnt           : Bitfield Pos 0, 3 Bits
      +0x000 Value            : Uint4B
(...)
   +0x218 MmProcessLinks   : struct _LIST_ENTRY, 2 elements, 0x8 bytes
      +0x000 Flink            : Ptr32 to
      +0x004 Blink            : Ptr32 to
   +0x220 ModifiedPageCount : Uint4B
   +0x224 Flags2           : Uint4B
   +0x224 JobNotReallyActive : Bitfield Pos 0, 1 Bit
(...)
```

Figura 14: Estrutura de um bloco *EPROCESS* (SCHUSTER, 2007).

```
kd> dt _PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
   +0x003 SpareBool        : UChar
   +0x004 Mutant           : Ptr32 Void
   +0x008 ImageBaseAddress : Ptr32 Void
   +0x00c Ldr              : Ptr32 _PEB_LDR_DATA
   +0x010 ProcessParameters : Ptr32 _RTL_USER_PROCESS_PARAMETERS
   +0x014 SubSystemData    : Ptr32 Void
   +0x018 ProcessHeap      : Ptr32 Void
   +0x01c FastPebLock      : Ptr32 Void
   +0x020 FastPebLockRoutine : Ptr32 Void
   +0x024 FastPebUnlockRoutine : Ptr32 Void
   +0x028 EnvironmentUpdateCount : Uint4B
   +0x02c KernelCallbackTable : Ptr32 Void
   +0x030 SystemReserved   : [2] Uint4B
   +0x038 FreeList         : Ptr32 _PEB_FREE_BLOCK
   +0x03c TlsExpansionCounter : Uint4B
   +0x040 TlsBitmap        : Ptr32 Void
   +0x044 TlsBitmapBits    : [2] Uint4B
   +0x04c ReadOnlySharedMemoryBase : Ptr32 Void
   +0x050 ReadOnlySharedMemoryHeap : Ptr32 Void
   +0x054 ReadOnlyStaticServerData : Ptr32 Ptr32 Void
   +0x058 AnsiCodePageData : Ptr32 Void
   +0x05c OemCodePageData  : Ptr32 Void
   +0x060 UnicodeCaseTableData : Ptr32 Void
   +0x064 NumberOfProcessors : Uint4B
   +0x068 NtGlobalFlag     : Uint4B
   +0x070 CriticalSectionTimeout : _LARGE_INTEGER
   +0x078 HeapSegmentReserve : Uint4B
   +0x07c HeapSegmentCommit : Uint4B
   +0x080 HeapDeCommitTotalFreeThreshold : Uint4B
   +0x084 HeapDeCommitFreeBlockThreshold : Uint4B
   +0x088 NumberOfHeaps    : Uint4B
   +0x08c MaximumNumberOfHeaps : Uint4B
   +0x090 ProcessHeaps     : Ptr32 Ptr32 Void
   +0x094 GdiSharedHandleTable : Ptr32 Void
   +0x098 ProcessStarterHelper : Ptr32 Void
   +0x09c GdiDCAttributeList : Uint4B
   +0x0a0 LoaderLock       : Ptr32 Void
   +0x0a4 OSMajorVersion   : Uint4B
   +0x0a8 OSMinorVersion   : Uint4B
   +0x0ac OSBuildNumber    : Uint2B
   +0x0ae OSCSDVersion     : Uint2B
   +0x0b0 OSPlatformId     : Uint4B
   +0x0b4 ImageSubsystem   : Uint4B
   +0x0b8 ImageSubsystemMajorVersion : Uint4B
   +0x0bc ImageSubsystemMinorVersion : Uint4B
   +0x0c0 ImageProcessAffinityMask : Uint4B
   +0x0c4 GdiHandleBuffer  : [34] Uint4B
   +0x14c PostProcessInitRoutine : Ptr32
   +0x150 TlsExpansionBitmap : Ptr32 Void
   +0x154 TlsExpansionBitmapBits : [32] Uint4B
   +0x1d4 SessionId        : Uint4B
   +0x1d8 AppCompatInfo    : Ptr32 Void
   +0x1dc CSDVersion       : _UNICODE_STRING
```

Figura 15: Estrutura de um *Process Environment Block* (*PEB*) *(SCHUSTER, 2006).*

2.4.3 *Pagefile*

Memória RAM é um recurso finito, então, quando a mesma está em seu limite, o *Windows* passa a alocar os processos com menor uso em arquivos no HD e, com isso, o sistema consegue ter

mais espaço livre na *RAM*, gerenciando os processos mais ativos (MICROSOFT, 2012d). Esses arquivos em HD que estão, temporariamente, armazenando processos da RAM são conhecidos como *pagefile* e os mesmos não devem ser esquecidos em uma análise do *dump* por se tratarem de uma extensão daquela.

2.5 Chamada a API do *Windows*

Application programming interface do *Windows* (APIW), é um sistema de interface de programação a nível de usuário para a família de sistemas operacionais *Windows* (RUSSINOVICH; SOLOMON; IONESCU, 2012). Este contém um conjunto de funções que possibilita aplicações a explorarem os recursos que o *Windows* oferece (MICROSOFT, 2010). São essas funções, como mostra a figura 16, responsáveis pela "ponte" entre o "modo usuário" e o "modo *kernel*" (RUSSINOVICH; SOLOMON, 2005). Essas APIs, que estão disponíveis ao usuário, são conhecidas como *Win32 API* e fornece um conjunto de serviços através de funções residentes em *DLLs* [3], enquanto que as APIs nativas são serviços oferecidos pelo *kernel* do sistema (SHARIF et al., 2008).

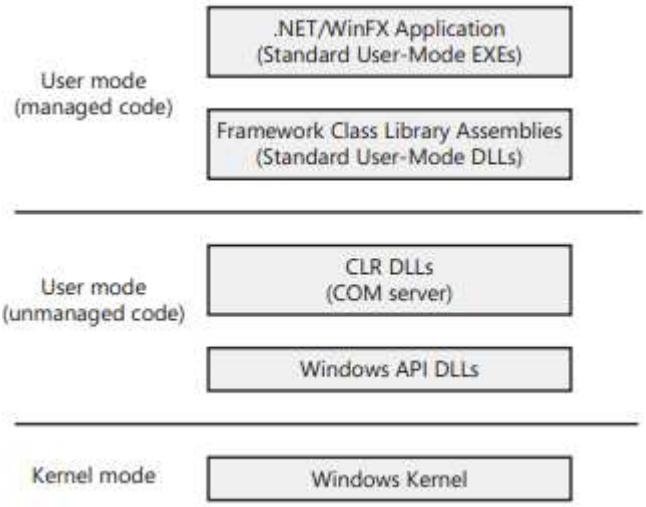


Figura 16: API (RUSSINOVICH; SOLOMON; IONESCU, 2012)

A APIW é composta por milhares de funções que podem ser subdivididas em categorias (MICROSOFT, 2010):

- *Administration and Management*: Conjunto de funções que permite a instalação e/ou configuração de programas;
- *Diagnostics*: Conjunto de funções que permite realizar correções de problemas de programas ou de sistema, além de monitorar a performance;
- *Graphics and Multimedia*: Conjunto de funções que interage com texto, áudio e vídeo;

[3]Dynamic-link library (DLL), são arquivos executáveis que atuam como bibliotecas de funções (DLLS, 2010). Portanto qualquer processo que faz uso de APIs do *Windows* também faz uso de DLLs (MICROSOFT, 2012a).

- *Networking*: Conjunto de funções que permite que aplicações se comuniquem com diferentes computadores na rede;
- *Security*: Conjunto de funções que realizam a autenticação do logon, acessos privilegiados, proteção dos objetos compartilhados e etc;
- *System Services*: Conjunto de funções que fornece à aplicação acesso aos recursos do sistema (*memória*, *drivers*, *threads* e etc);
- *Windows User Interface*: Conjunto de funções que permite que aplicações possam criar janelas de saída como meio de interface.

Como visto, qualquer programa simples faz uso de funções da API. Através da identificação de todas essas chamadas, pode-se traçar um comportamento do *malware* (WANG et al., 2009). Algumas arquiteturas de geração de assinaturas, como a *Medusa* (NAIR et al., 2010), geram a saída baseadas somente em um conjunto de APIs estraídas, *critical APIs (CAPIs)*, onde esse é o conjunto das k funções chamadas com mais frequência (no artigo da arquitetura *Medusa* k=30).

2.6 Conclusão

Este capítulo apresentou como funciona o sistema de arquivos *NTFS*, utilizado atualmente em ambientes *Windows*, e qual a importância de análisar informações desse sistema para se obter informações de, por exemplo, quais arquivos foram modificados/apagados/criados pelo *malware.* Sobre os Registros do *Windows* a respectiva seção abordou o quão importante são as informações armazenadas em arquivos *hives* e como os vírus a utilizam para se manterem no sistema.

Foi mostrado também que qualquer aplicação sendo executada mantém informações importantes na memória RAM, logo, a realização do *dump* para análise mostra-se como mais uma fonte de dados que colabora na criação de assinaturas. Por último pode-se traçar um perfil comportamental do *malware* utilizando, também, as funções da API que foram invocadas pelo vírus, tendo em vista que o uso de recursos oferecidos pelo *Windows* terá que ser feito utilizando, obrigatoriamente, essas funções.

Capítulo 3

Introdução aos *malwares*

3.1 Introdução

Este capítulo apresentará os tipos de *malwares* existentes, será abordado as técnicas utilizadas por criadores de vírus para dificultar a análise por especialistas e como as empresas de Anti-Vírus lidam com cada um.

É importante salientar que, de forma geral, todos os vírus mantém o mesmo padrão de comportamento, ou seja, realizam a mesma sequência de atividades que são (KRUEGEL et al., 2005):

1. Escanear (*Scan*): O *malware* busca por vulnerabilidades no sistema;
2. Comprometer (*Compromise*): Ao obter sucesso no escaneamento o vírus explora a falha, comprometendo o sistema;
3. Replicar (*Replicate*): Após se instalar, o vírus busca se replicar para outros ambientes (o máximo possível), criando novos vírus que reiniciarão o ciclo: Escanear, Comprometer e Replicar.

Antes do aprofundamento nos tipo de *malware*, faz-se necessária a definição de alguns termos:

- Assinatura/*fingerprint*: técnica que busca extrair padrões dos *malwares* afim de, posteriormente, conseguir identificá-los. Atualmente existem duas abordagens na extração de assinaturas para vírus (ALAZAB et al., 2010):
  - *Pattern/string-matching*: técnica na qual é extraído um conjunto de *strings* do *malware,* onde são criadas as regras para validação do vírus. Essas condições são tidas como únicas para a detecção do *malware* e, para fazer tal identificação, é feita uma busca por padrão (*pattern-matching*). A grande vantagem está em detectar facilmente os vírus conhecidos, porém tal técnica falha ao se levar em consideração novos *malwares* (exploração de *0-day*[1]*;*

[1]Termo que se dá a novas falhas descobertas.

  - *Anomaly-based*: técnica que leva em consideração o comportamento do vírus afim que seja criada a assinatura. Essa abordagem se utiliza portanto, dos conhecimentos sobre comportamento malicioso.

- *Anti-debugging*: técnica que dificulta ou compromete o processo de *debug*;
- *Anti-disassembly*: técnica que compromete o processo de *disassembly*;
- *Obfscation*: técnica que compreende o comprometimento do *disassembly* e a idéia de dificultar a criação de assinaturas, ou seja, ofuscar o binário;
- *Anti-VM*: técnica para detectar e/ou comprometer análises em máquinas virtuais (VM - Virtual Machine) (BRANCO; BARBOSA; NETO, 2012).

3.2 Tipos de *Malwares*

No que se refere aos tipos de *malwares* existentes, pode-se classificá-los quanto ao seu tipo. Obedecendo essa classificação,conclui-se que há três grandes conjuntos:

- *Static Malware*;
- *Polymorphic Malware*; e
- *Metamorphic Malware*.

Neste capítulo todos serão abordados afim de tornar clara: estrutura do código, vantagens, desvantagens (as duas últimas são do ponto de vista do criador de *malwares*) e como empresas de anti-vírus combatem tais pragas.

3.2.1 *Static Malware*

Tido como o tipo de vírus mais simples, o *Static Malware* não é usualmente desenvolvido entre os autores, isso por ser de fácil detecção e uma vez detectado o mesmo estará, para sempre, comprometido. Como o próprio nome sugere esse *malware* é considerado estático por manter o mesmo padrão estrutural; ou seja, não sofre com sistemas de encriptação, *anti-debugging*, *anti-disassembly* e etc, ou em outras palavras, não gera mutações na replicação.

Entre suas vantagens, está a facilidade de sua criação devido à uma estrutura simples (somente o código bruto referente ao que o vírus foi projetado para fazer). Por outro lado, dentre as desvantagens está a facilidade na criação de assinaturas; uma vez que essa assinatura esta em um banco de dados de anti-vírus (AV), esse nunca mais deixa de ser detectado. Esse é um dos motivos pelo qual não há muitos *samples* desse conjunto.

Entre as empresas de AV, a criação automática de assinaturas consiste em uma simples extração do *hash* (MD5, por exemplo) para porterior detecção por meio do processo de *pattern matching*, ou, *string matching* (CLAMAV, 2010).

3.2.2 *Polymorphic Malware*

São classificados como *Polymorphic Malware* os vírus que possuem a característica de auto-replicação, com criação de mutações, sem alterar sua funcionalidade. Esse grupo de *malwares* contém um sistema de encriptação-decriptação embutido ao código malicioso. Existem diversas chaves para a encriptação-decriptação, sendo que a cada auto-replicação o *malware* utiliza uma delas, criando então um novo *malware* com uma nova chave de encriptação.

Como dito, a estrutura desse vírus é composta do código fonte malicioso e um sistema de encriptação-decriptação (a figura 17 ilustra essa estrutura), o código malicioso mantém-se encriptado e somente quando o mesmo for executado ele é decriptografado e alocado na memória RAM do sistema (KAUSHAL; SWADAS; PRAJAPATI, 2012).

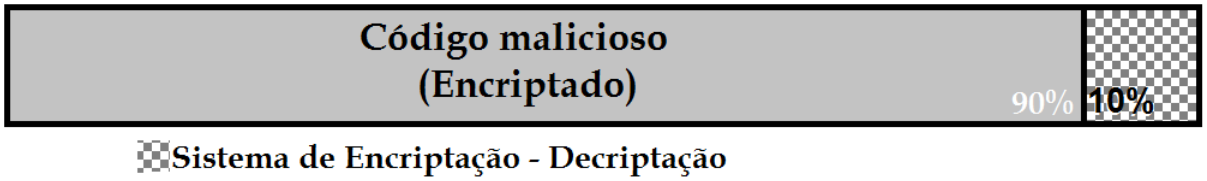


Figura 17: Estrutura de um *Polymorphic Malware*. (KAUSHAL; SWADAS; PRAJAPATI, 2012)

A grande vantagem desse vírus está em se conseguir criar um grande leque de mutações do mesmo *malware*, atrapalhando sua identificação junto às empresas de anti-vírus. Isso ocorre, por exemplo, porque se a empresa de anti-vírus analisa e extrai a assinatura do vírus “A”, encriptado pela chave “K1”, essa mesma assinatura não consegue detectar o mesmo vírus encriptado com a chave “K2” (o vírus “A” se replica utilizando uma nova chave, no caso a K2). A figura 18 ilustra a replicação desses vírus e a vantagem da variabilidade criada.

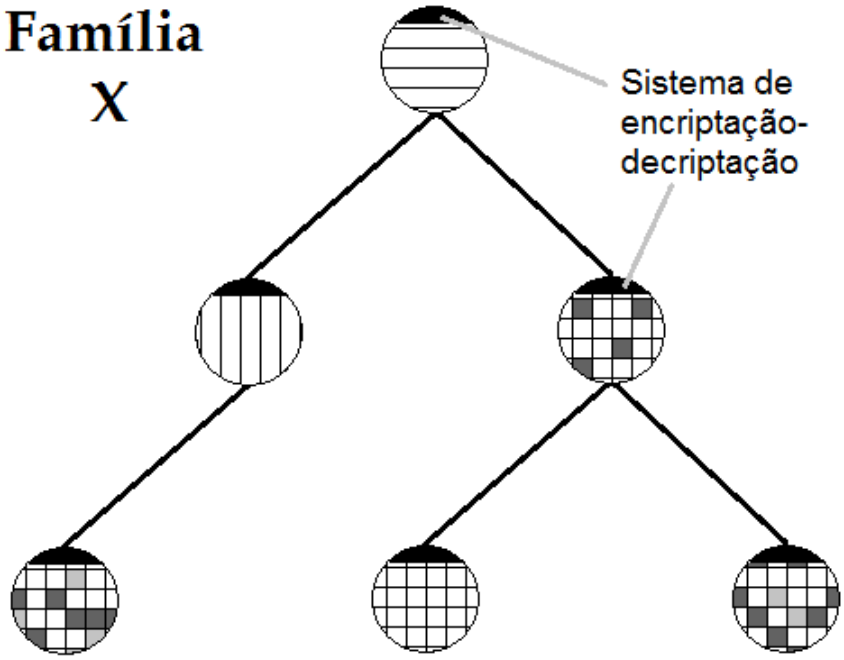


Figura 18: Árvore genealógica de um *Polymorphic Malware*.

Para se criar uma única assinatura capaz de identificar toda uma "família", de *Polymorphic Malware*, especialistas se fundamentam exclusivamente no "Sistema de Encriptação-Decriptação" (referente aos 10% da figura 17), por concluirem que independente da versão do *malware* o sistema se mantém o mesmo, como ilustra a figura 18 onde mostra *malwares* com encriptações distintas mas com o bloco se mantendo idêntico. Esse "bloco" de encriptação-decriptação é tido como a principal desvantagem dos *Polymorphics Malwares*, pois esse mantém-se inalterado (KAUSHAL; SWADAS; PRAJAPATI, 2012).

Para a criação de assinaturas para esse tipo de vírus aplica-se a mesma técnica usada no *Static Malware*, *hash*, que, nesse caso, será criado sobre o bloco de encriptação-decriptação (CLAMAV, 2010).

Por último, é importante registrar que existem, em menor grau, *Polymorphic Malwares* que também mutam seu bloco de encriptação-decriptação, os quais são alvo da proposta desse trabalho, descrita no Capítulo 4. Também existem AVs mais sofisticados que permanecem analisando esses *malwares* a espera do momento em que o mesmo se aloque na memória, e nesse momento o AV extrairá a assinatura do *sample* (YOU; YIM, 2010). Isso porque logo antes de ser carregado na RAM o código malicioso encriptado será decriptado e sobre esse código decriptado a assinatura é criada, logo esse *fingerprint* independe da chave utilizada para detecção do *malware*.

3.2.3 *Metamorphic Malware*

Tido como o mais complexo entre os demais, os vírus classificados como *Metamorphic Malwares* são melhores elaborados e menos passíveis de detecção. Assim como *Polymorphic Malware*, esse

tipo de vírus consegue se replicar alterando sua estrutura (criando mutações) e mantendo sua funcionalidade. O que distingue um *Metamorphic* de um *Polymorphic Malware* é que esse utiliza técnicas de encriptação para gerar mutações (KAUSHAL; SWADAS; PRAJAPATI, 2012), enquanto que aquele utiliza-se de técnicas como: *anti-debugging, anti-disassembly, obfuscation* e *anti-VM* (BRANCO; BARBOSA; NETO, 2012).

O principal objetivo por trás do uso dessas técnicas é impedir e/ou dificultar a análise por parte dos especialistas e por conseguinte evitar a criação de assinaturas. Tal objetivo é conquistado nos *Metamorphic Malwares* por serem capazes de criar códigos totalmente diferentes a cada replicação, sem que haja trechos iguais entre quaisquer variações. A figura 19 ilustra essa situação, onde pode-se notar que a cada mutação, uma variação sem qualquer igualdade com seu antecessor é gerada (KAUSHAL; SWADAS; PRAJAPATI, 2012).

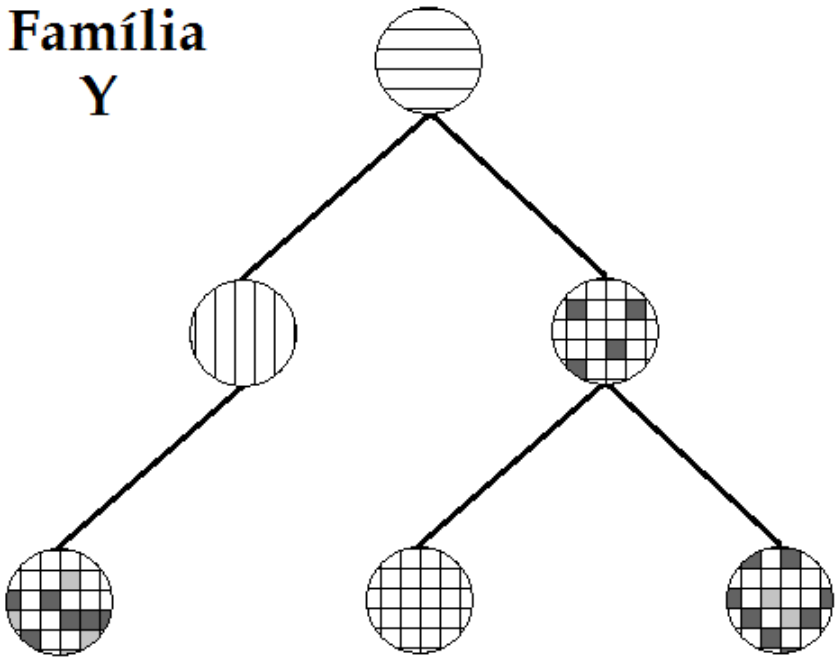


Figura 19: Árvore genealógica de um *Metamorphic Malware*.

A estrutura (figura 20) desse tipo de *malware* é composta pelo código malicioso e um Sistema de Transformação, o qual é composto por todas as técnicas citadas, sendo dele a responsabilidade de tornar o código inédito a cada mutação, tornando a análise bastante complexa (KAUSHAL; SWADAS; PRAJAPATI, 2012).

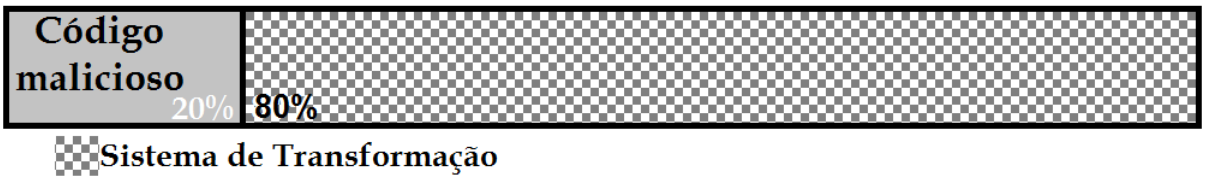


Figura 20: Estrutura de um *Metamorphic Malware*. (KAUSHAL; SWADAS; PRAJAPATI, 2012)

A caracterítisca de variabilidade entre as versões do *malware* e sua consequente dificuldade de análise são as principais vantagens desse vírus em termos de eficácia. No entanto, por maior que seja a diversidade existente, a funcionalidade do sistema será a mesma, e consequentemente o seu comportamento também. O que facilida a criação de assinaturas baseadas em comportamento, uma das premissas abordadas no Capítulo 4.

3.2.3.1 *Anti-Debugging*

Como citado anteriormente, técnica de *Anti-Debugging* é utilizada por autores de *malwares* para dificultar e/ou comprometer o processo de engenharia reversa realizado por especialistas. Como "mecanismo de defesa", o vírus fica a todo instante verificando a incidência de *debugging*. Em caso afirmativo o mesmo pode, por exemplo, se auto-destruir. Algumas técnicas utilizadas para identificação de análises de *debugger* são:

- Checagem do campo *boolean* "*BeingDebugged*" da estrutura *PEB* (figura 15, linha 4): se este for *TRUE*, então está sendo analisado por algum *debugger* (MICROSOFT, 2012b);
- Fazer *auto-debugging*. Essa técnica é conhecida como *self-debugging* e consiste em envolver ao processo um *debugger* para que realize um *debug* de si mesmo, o que evita que o *malware* sofra *debugging* por especialistas, tendo em vista que todo processo não pode passar por mais de um *debug* ao mesmo tempo (FERRIE, 2010);
- *FindWindow*. Nessa técnica o processo simplesmente é de verificação de janelas de *debuggers* abertas (BRANCO; BARBOSA; NETO, 2012);
- Na técnica *Sofware BreakPoint Detection*, o processo verifica se algum *BreakPoint* foi colocado, o que é possível pois, no *BreakPoint*, a execução do processo é interrompida e o controle é passado para o *debugger* (YASON, 2007).

3.2.3.2 *Anti-Disassembly*

Disassembly é uma técnica no qual o *software* (*disassembler*) irá percorrer por todo o binário do processo e por fim extrair o código *assembly*. Disso, o especialista em *malwares* pode analisar melhor como o vírus funciona e por fim traçar uma assinatura para o mesmo. De modo geral os *disassemblers* trabalham com duas abordagens: *linear sweep* e *recursive traversal*.

- *Linear sweep*: o *disassembler* inicia no primeiro *byte* do binário do processo a ser analisado e assim vai até o final, decodificando uma instrução após a outra, porém a principal desvantagem desse tipo de abordagem é que dados/comandos, que nunca serão executados, inseridos no meio do código irão ser interpretados e isso acabará gerando algum tipo de "ruído", ou seja, o *disassembly* poderá ocorrer de forma errada.
- *Recursive traversal:* segue o fluxo do programa, logo esse não decodifica dados que não são executados, no entanto, a principal desvantagem dessa abordagem é que nem sempre pode-se inferir qual o fluxo do processo e isso acaba gerando partes do binário sem sofrer *disassembled* (BRANCO; BARBOSA; NETO, 2012) (KRUEGEL et al., 2004).

Algumas das técnicas utilizadas para evitar o *disassembly* são a *Garbage bytes* (pode influenciar tanto na vertente *linear sweep* quanto na *recursive traversal* (HARBOUR, 2011)), *Fake Conditional Jumps* e etc. Sem se aprofundar em como tais técnicas funcionam pode-se observar no algoritmo 3.1 que tudo que estiver após a linha 1 e antes da linha 3 não será executado (*garbage bytes*) porém esses dados influenciam no momento do *disassembly* (YASON, 2007).

Algoritmo 3.1 Técnica *Garbage Bytes* (BRANCO; BARBOSA; NETO, 2012).

```
1          jmp .destination
2          db  6Ah ; garbage bytes
3  .destination:
4          pop eax
5          ... ; restante do código
```

3.2.3.3 *Obfuscation*

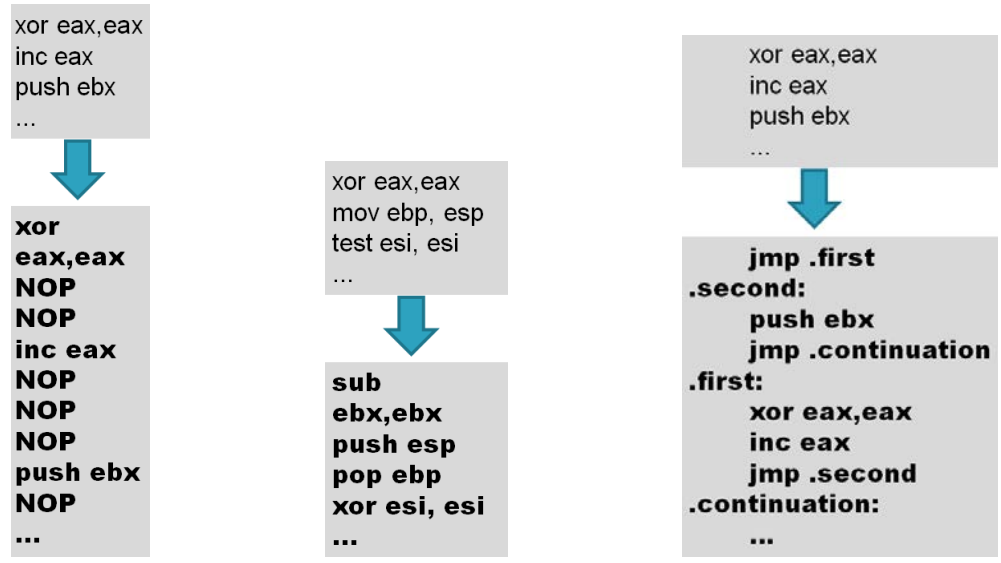
Essa técnica tem como objetivo ofuscar o que ocorre no código e assim dificultar a criação de assinaturas pois uma única mudança no binário irá alterar o *fingerprint* que é criado em análises automáticas. Algumas técnicas utilizadas são: *NOP Sequence, Instruction Substitution, Register Reassignment, Code Transposition* e etc.

- *NOP Sequence*: Técnica em que insere-se no código alguns comandos NOP afim de tornar a assinatura do mesmo diferente da anterior. O comando NOP não modifica em nada a funcionalidade do código, essa inserção é considerada como "inserção de código morto" (a figura 21a mostra um código antes e depois da inserção de NOPs, pode-se notar quem em sistemas automatizados de extração de assinaturas esses códigos terão saídas diferentes). Alguns outros exemplos de "código morto" são: "*sub ax, 0*", "*mov ax, ax*", etc (CHRISTODORESCU; JHA, 2003);

- *Instruction Substitution:* Nessa técnica é utilizado um dicionário de instruções equivalentes para serem substituídas do código original, por exemplo, trocar um "*jmp*" por "*xor eax, eax* $\rightarrow$ *jz*" ou trocar um "*mov*" por "*push* $\rightarrow$ *pop*" (BRANCO; BARBOSA; NETO, 2012).

- *Register Reassignment*: Os registradores serão trocados por equivalentes, por exemplo, todos os registradores "*eax*" serão substituídos por "*ebx*" (CHRISTODORESCU; JHA, 2003). A figura 21b exemplifica o uso dessa técnicas em conjunto com *Instruction Substitution*, vale ressaltar que as assinaturas serão distintas após sofrerem tais mudanças;

- *Code Transposition*: Essa técnica consiste em simplesmente embaralhar todas as instruções e manter o comportamento do processo através de inserções de *jumps* no intuíto de fazer com que esse comando reestabeleça a ordem do fluxo de execução das instruções tornando-o idêntico ao original (antes de ser embaralhado) (YOU; YIM, 2010). A figura 21c demonstra a aplicação dessa técnica.



(a) *NOP Sequence.*

(b) *Instruction Substitution + Register Reassignment.*

(c) *Code Transposition.*

Figura 21: Técnicas de *Obfuscation*.

3.2.3.4 *Anti-VM*

A grande maioria de laboratórios de *malwares* utilizam máquinas virtuais (VM) para analisar seus vírus, isso porque esse ambiente permite maior segurança no controle de variáveis do sistema, maior facilidade na análise e maior rapidez na criação de novas máquinas para novas análises. Com isso os autores de vírus criaram uma técnica na qual seus programas checam se estão sendo executados em máquinas virtuais e em resultados positivos o mesmo pode, por exemplo, se auto-destruir. Essa técnica foi criada afim de dificultar a análise do mesmo por especialistas tendo em vista que quando o *malware* estiver sendo executado em uma VM existe uma alta probabilidade do mesmo estar passando por uma análise.

Uma forma bem simples de fazer essa checagem é executar um comando a nível de *kernel*. Caso o *malware* esteja em uma máquina real será retornado a ele uma *exception* pois somente o S.O. pode invocar comandos a nível de *kernel*. Já em uma VM essa *exception* não será retornada, isso porque o *software* que controla a comunicação entre a máquina virtual e o S.O. da máquina real fará o tratamento da chamada da função ao S.O. da máquina real e esse então se encarregará de executar a função a nível de *kernel* (BACHAALANY, 2011).

3.3 Conclusão

É possível observar por meio desse capítulo como os *malwares* funcionam e algumas das técnicas que utilizam para burlar o sistema de criação automática de assinaturas do tipo *pattern-matching*. É notável também que uma pequena parte de *Polymorphic Malwares* e todos os *Metamorphic*

Malwares têm a capacidade de auto-mutação, possibilitando que uma assinatura criada para dada versão não seja capaz de identificar a versão seguinte e com isso, torna-se muito difícil a criação de *fingerprints*.

Com esse problema e tendo em vista a idéia de criação de assinaturas, uma "família Y" (figura 19) com "Z" versões criadas deve ter "Z" assinaturas extraídas; ou seja, cada versão teria sua própria assinatura, tornando essa prática inviável. No capítulo 4 é proposta uma solução para essa dificuldade.

Capítulo 4

Proposta de Arquitetura e Protocolo para Geração Automática de Assinaturas de *Malwares*

4.1 Introdução

Este capítulo apresentará a arquitetura e o protocolo propostos para a geração automática de assinaturas de *malwares*, será abordado seu funcionamento e composição.

Seu propósito é a extração de informações capazes de identificar famílias inteiras de *Metamorphic* e *Polymorphic Malwares,* sem que encriptações ou técnicas como *anti-debugging, anti-disassembly, obfuscation* ou *anti-VM* interfiram no resultado final.

Como visto no capítulo anterior existem dois tipos de assinaturas (*pattern-matching* e *anomaly-based*). Com a intenção de retirar o máximo de informações do *malware,* o protocolo proposto aborda uma extração híbrida, ou seja, tanto a vertente de assinaturas baseadas em *pattern-matching,* quanto as baseadas em *anomaly-based* são efetuadas.

4.2 Arquitetura Proposta

A arquitetura desse sistema é composta por seus grupos (todos podem ser visualizados na figura 22):

1. Computadores responsáveis por manter VMs executando;
2. Computadores responsáveis por executar os *malwares* que se recusaram a serem executados nas VMs;
3. Dois bancos de dados:
   (a) Armazenamento do *sample* em sua chegada (BD1 da figura 22); e

(b) Armazenamento das informações extraídas do *sample* para geração da assinatura (BD2).

4. Elemento para o gerenciamento do recebimento dos *malwares* (computador (I), figura 22);
5. Elemento para a implementação do *Scheduler*; e
6. Elemento para a geração da assinatura (computador (II));

O implementação de todos os itens citados acima, juntamente com a explicação de funcionamento, é abordada na seção 4.3.

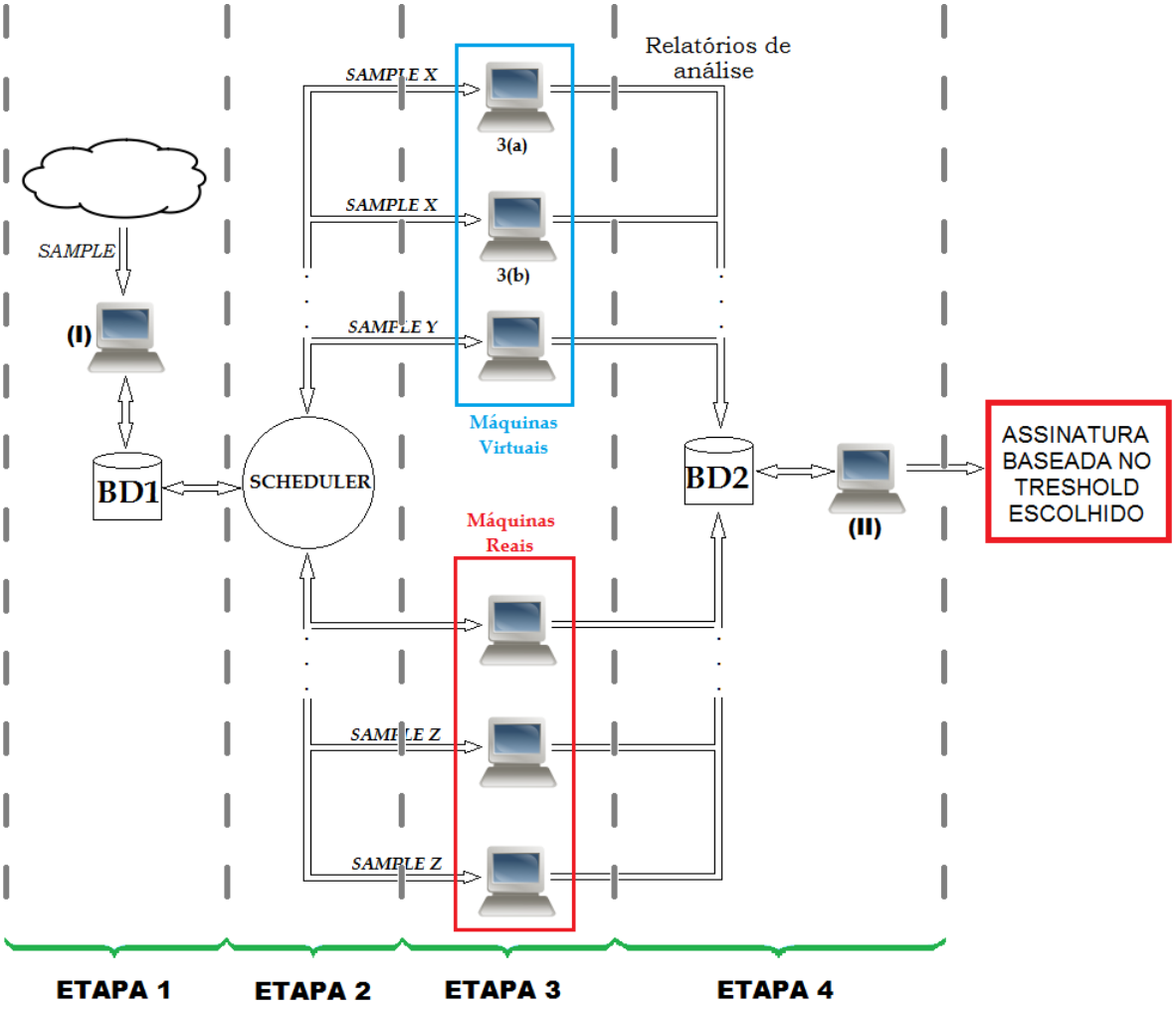


Figura 22: Etapas da arquitetura.

4.3 Protocolo Proposto

Nessa seção é abordado o protocolo proposto a ser implementado na arquitetura do item anterior. Tal protocolo é composto basicamente por quatro etapas: Conferência dos *samples*, Alocação para análise, Análise dos *samples* e extração de assinatura (todas as etapas serão abordadas com maiores

detalhes). O término de todas etapas levam a extração de informações do *sample* onde culmina com a geração da assinatura.

Com isso pretende-se conseguir o entendimento do completo funcionamento do sistema.

4.3.1 Etapa 1 - Conferência dos *samples*

A etapa 1 possui um sistema (computador (I) da figura 22) responsável por gerenciar o recebimentos de todos os vírus que chegam pela rede.

De início, tal computador verifica se o *malware* recebido já existe no BD1 (figura 22), caso esse não seja encontrado conclui-se que se trata de um novo *malware.* Caso seja existente o sistema verifica se o vírus já foi executado ou se ainda está na fila para alocação, na ocasião da primeira opção, o vírus simplesmente será ignorado (excluído do sistema), senão tal *malware* terá sua prioridade de execução incrementada no BD1. Para novos casos de vírus o sistema o armazena no banco de dados 1 com sua devida prioridade (o funcionamento do sistema de prioridades será explicado na seção 4.3.2).

É importante citar que o BD1 possui, no mínimo, cinco colunas:

1. *Flag:* indica se o *malware* já foi para análise (*flag* = 1), ou se ainda está na fila de espera (*flag* = 0);
2. *Path*: indica onde o *malware* é armazenado no sistema;
3. Tempo de Vida (TV): indica quantos ciclo de execução se passaram sem a alocação do *sample*; ou seja, toda vez que o *scheduler* consulta o banco para enviar um vírus à análise, todos aqueles não selecionados têm o conteúdo dessa coluna incrementado;
4. Número de Repetições (NR): após o computador (I) concluir que o *sample* existe no sistema, é realizada uma consulta à coluna "*Flag*":
   (a) se *flag*=0 então NR = NR + 2;
   (b) se *flag*=1 então o *malware* será ignorado (excluído do sistema).
5. Prioridade (P): indicador no qual o *scheduler* se fundamenta para a escolha do *sample* a ser alocado para análise. A prioridade (P) será definida por: P = TV + NR.

4.3.2 Etapa 2 - Alocação para análise

A etapa 2, ilustrada pela figura 22, é composta por um *scheduler* responsável por manter as máquinas de análise sempre ocupadas, alimentando-as com novos *samples*, gerenciar casos de vírus que se recusam a serem executados em máquinas virtuais (utilizam técnicas de *anti-VM*), ou seja, enviar para máquinas reais e atualizar o sistema de Prioridades.

O *scheduler* sempre estará consultando o BD1 para adquirir novas amostras de *malwares* a serem analisadas. A proposta geral desse protocolo da etapa 2 será embasada no trabalho de Branco e Shamir (2010), que pode ser adaptado a essa arquitetura.

Baseando-se no "*Priority Scheduling Algorithm*", para a escolha de qual vírus é alocado para análise, tem-se o seguinte protocolo de prioridade:

- Todos os vírus novos, não existentes no BD1, recebem prioridade igual a 0 (TV e NR igual a zero);
- Um *malware* aumenta seu nível de prioridade (P = TV + NR) se:
  - Passar por um ciclo de execução sem ser alocado para análise (seu TV é incrementado em uma unidade pelo *scheduler*);
  - O sistema receber um vírus idêntico a algum outro que presente no BD1, logo isso pode representar que tal *malware* está em destaque na rede mundial de computadores (seu NR será incrementado em 2 unidades pelo computador (I)).

Para ilustrar melhor a tarefa do *scheduler,* vamos exemplificar o processo que será realizado para decidir qual *sample* deverá ir para análise:

1. Na coluna prioridade, do BD1, será realizado um *sort;*
2. Dentre os resultados obtidos do processo anterior seleciona-se o de maior prioridade. Em caso de empate, realiza-se um *sort* na coluna número de repetições, do BD1, somente entre os que estão com a prioridade empatadas;
3. Dentre os resultados obtidos do processo anterior seleciona-se o que possui maior número de repetições. Em caso de empate, realiza-se um *sort* na coluna tempo de vida, do BD1, somente entre os que estão com o número de repetições empatadas;
4. Dentre os resultados obtidos do processo anterior seleciona-se o que possui maior tempo de vida e, em caso de empate, fica-se com o primeiro da lista;
5. Altera-se a *flag* do *malware* escolhido para 1;
6. Incrementa-se, em uma unidade, a coluna TV de todos os *malwares* com *flag*=0.

As máquinas físicas com máquinas virtuais instaladas em seu sistema serão responsáveis por:

- Criar as máquinas virtuais a partir de uma imagem pré-estabelecida (imagem mestra);
- Sinalizar ao *scheduler* se há um par de VMs ociosas;
- Sinalizar ao *scheduler* se o *malware* se recusou a ser executado, ou seja, o *malware* possui sistema de *Anti-VM*;
- Receber as informações de cada VM sobre os *malwares* que estão sendo analisados (informações essas que irão compor o relatório de saída);
- Destruir a máquina virtual após o término da análise;
- Encaminhar os relátorios ao banco de dados 2 (BD2 da figura 22), já com as saídas da VM responsável pela análise do comportamento e da VM responsável pela análise do código unidos ao mesmo relatório.

4.3.3 Etapa 3 - Análise dos *samples*

A etapa 3 (figura 22), que contém duas subetapas (3(a) e 3(b) da figura 23), responsável pela análise do *malware*, que ocorrerá em máquinas virtuais ou reais dependendo de cada caso. É interessante ressaltar que o mesmo vírus sempre será enviado a duas máquinas distintas, sendo que uma fará análise comportamental e, outra, uma análise do código.

Na subetapa 3(a) (figura 23a), o *sample* em questão será executado por exatamente um minuto e durante esse tempo estará sob análise os seguintes parâmetros:

- Sistema de Arquivos: dessa análise deve-se extrair informações de todos os arquivos ou diretórios criados, alterados ou deletados do sistema. Todo incidente terá as *paths* de tais arquivos armazenadas em relatório juntamente com o conteúdo das informações criadas ou modificadas, quando for o caso;
- Registros do *Windows*: a observação desses dados deverá mostrar quais chaves/values do registro foram alterados ou deletados. Em caso de alterações, as informações adicionadas também serão capturada;
- *Dump* da RAM: ao final dos sessenta segundos que o vírus estará em execução o *dump* da RAM será realizado para uma posterior análise, que será realizada em comparação com o *dump* da RAM da imagem mestra (da mesma maneira que existe uma imagem mestra para a criação de todas as VMs existirá também uma *dump* mestra da RAM já armazenada para realizar tais comparações). Essa confrontação ajudará a concretizar a análise do *dump* de maneira mais ágil;
- Chamadas a API: todas as funções que fazem parte da API do *Windows* que forem invocadas serão armazenadas e, junto a ela, será registrado a quantidade de vezes que tal função foi chamada. Após o tempo de execução do *malware* terminar, deverá ser extraída as k funções mais frequentes. Além de monitorar as funções também será monitorado o que tal função fará, por exemplo: se for invocada a função *DeleteFile* deve-se registrar que tal função foi chamada e qual arquivo será deletado; se for invocada a função *CreateFile* deve-se registrar foi chamada e os parâmetros enviados: nome do arquivo a ser criado e *path*. Quanto ao conteúdo do arquivo, esse será enviado pela função *WriteFile*.

A princípio pode parecer estranho monitorar todos esses parâmetros, sendo que para qualquer atividade ocorrer alguma função da API deve ser invocada, ou seja, bastava-se analisar as chamadas a API e o objetivo seria conquistado, porém é importante salientar que existem técnicas conhecidas como *"API call obfuscation"* onde o uso dela pode comprometer a integridade de uma análise baseada exclusivamente em chamadas a API.

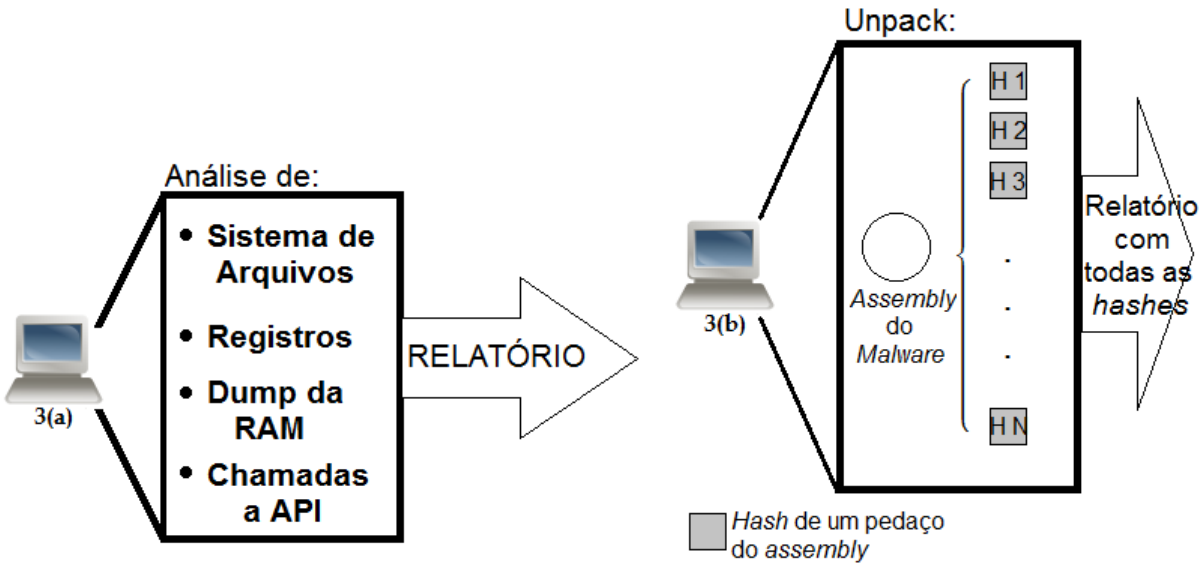
Diante disso, optou-se por uma análise mais abrangente e, por fim, os dados coletados individualmente e os das chamadas a API serão confrontados para se garantir uma extração de dados fidedigna.

Já na subetapa 3(b) (figura 23) será feito o *unpack*[1] do binário do *sample*, com o intuíto de se extrair o *assembly* e, com isso, dividir o fonte em pedaços (pequenos blocos de código), para que finalmente extraia-se *hash*es de cada um desses blocos.

[1]Técnica para extração do *assembly* do executável.

A vantagem dessa abordagem está em levar em consideração uma vizinhança menor entre as instruções de cada bloco e isso trará uma maior precisão na análise, principalmente no momento em que for calculado o grau de parentesco entre *malwares* (seção 4.3.4).

Após o término da etapa 3, o sistema presente na máquina real, gerenciando as VMs, ficará responsável por agrupar essas informações e enviar para armazenamento no BD2 (ver figura 22). Com isso, o *sample* X, por exemplo, tem agora suas informações extraídas e devidamente relatadas.



(a) Subetapa (a): Análise do comportamento do *malware.* (b) Subetapa 3(b): Análise do código do *malware.*

Figura 23: Subetapas do protocolo proposto.

4.3.4 Etapa 4 - Extração de assinatura

Para finalizar o protocolo de análise automática de *malwares,* outro sistema presente na etapa 4 (figura 22, computador (II)) irá consultar os relatórios armazenados no BD2 e calcular o grau de parentesco entre os *samples,* sempre que solicitado, com o *threshold* estabelecido. Todos esses cálculos serão armazenados em uma nova tabela, onde os *malwares* serão agrupados em conjuntos, de acordo com suas famílias (obedecendo o grau de parentesco). Com isso, pode-se extrair uma assinatura a partir do *threshold* estabelecido.

Um exemplo está em requisitar um *threshold* em 80 e como saída teriamos todos os conjuntos de *malwares* que são 80% semelhantes (levando-se em consideração tanto o comportamento quanto o código), desses conjuntos são extraídas as assinaturas que serão capazes de identificar todos os *samples* pertencentes ao conjunto do limiar pré-estabelecido. Esse grau de parentesco seria calculado utilizando-se conceitos de matriz de distância.

4.4 Conclusão

Este capítulo apresentou a arquitetura, composta por seis elementos, e o protocolo de quatro etapas que juntos formam o sistema proposto, cuja função é gerar automaticamente assinaturas de *malwares* com base na extração de dados dos vírus.

Capítulo 5

Conclusão e Trabalhos Futuros

5.1 Conclusão

Mesmo utilizando-se de técnicas como *anti-debugging, anti-disassembly, obfuscation* e *anti-VM* foi visto que os *malwares* mantém a sua funcionalidade e consequentemente seu comportamento, logo a abordagem de extrair a assinatura baseada em comportamento e em análise do código do *malware* se mostra muito mais eficiente, tanto na detecção de um conjunto maior de vírus com uma única assinatura quanto na diminuição de falsos positivos.

A arquitetura e o protocolo propostos se destacam por possuir os seguintes diferenciais:

- Criação da assinatura baseada de forma híbrida: comportamento do vírus e análise de código;
- Quatro parâmetros de análise para se traçar o perfil comportamental do *malware*:
  - Análise do sistema de arquivos;
  - Análise de modificação dos registros do *Windows*;
  - Análise do *Dump* da RAM;
  - Análise das chamadas a *API*.
- Assinatura capaz de identificar toda uma família de *malwares;*
- Extração de regras que podem ser implementadas em sistemas *IPSs;*

O protocolo proposto visa se destacar na identificação de *Metamorphic Malwares* e, em menor grau, na identificação de *Polymorphic Malware*s.

Outro ponto interessante desse protocolo é a capacidade em se conseguir extrair tanto assinatura quanto regras para implementação em *IPSs* baseada em *threshold* pré-estabelecido. Isso significa que vírus com graus de diferença não tão distantes (mesmo pertencentes a famílias diferentes) serão detectados com uma única assinatura.

5.2 Trabalhos Futuros

Como forma de levar esse trabalho adiante ficará como trabalhos futuros a implementação da arquitetura e protocolo; a inclusão de mais um parâmetro para análise: “Fluxo de dados da rede”; e a criação de um sistema de optimização a ser implementada nos bancos de dados.

Acrescentar o parâmetro de análise de dados da rede é de suma importância para tornar a arquitetura mais robusta pois dele poderemos extrair informações que o *malware* envia e recebe pela rede de computadores.

Já a implementação de um sistema de optimização será necessário pois esses irão ficar cada vez mais volumosos com a chegada de novos vírus, tornando então, a consulta cada vez mais lenta e esse pode se tornar fator limitante na velocidade de análise.

Referências Bibliográficas

ALAZAB, M. et al. Malware detection based on structural and behavioural features of api calls. 2010.

BACHAALANY, E. Detect if your program is running inside a virtual machine. 2011.

BOSCOVICH, R. D. *Microsoft Neutralizes Kelihos Botnet, Names Defendant in Case*. The Official Microsoft Blog, 2011. Disponível em: <http://blogs.technet.com/b/microsoft_blog/archive/2011/09/27/microsoft-neutralizes-kelihos-botnet-names-defendant-in-case.aspx>.

BRANCO, R. R. *IPS na intimidade*. [S.l.]: Linux Magazine, 2007.

BRANCO, R. R.; BARBOSA, G. N.; NETO, P. D. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. 2012.

BRANCO, R. R.; SHAMIR, U. Architecture for automation of malware analysis. 2010.

CARVEY, H. *Windows Forensic Analysis, Second Edition*. [S.l.]: Syngress, 2011. 89–155 p. ISBN 978-1-59749-422-9.

CARVEY, H. *Windows Registry Forensics*. [S.l.]: Syngress, 2011. ISBN 978-1-59749-580-6.

CARVEY, H.; ALTHEIDE, C. *Digital Forensics with Open Source Tools*. [S.l.]: Syngress, 2011. ISBN 978-1597495868.

CASEY, E. *Digital Evidence and Computer Crime*. [S.l.]: Academic Press, 2011. ISBN 978-0123742681.

CHRISTODORESCU, M.; JHA, S. Static analysis of executables to detect malicious patterns. 2003.

CLAMAV. *Creating signatures for ClamAV*. ClamAV, 2010. Disponível em: <http://www.clamav.net/doc/latest/signatures.pdf>.

CONSTANTIN, L. *Largest DDoS attack so far this year peaked at 45Gbps, says company*. Computer World, 2011. Disponível em: <http://www.computerworld.com/s/article/9222156>.

DAMBALLA. *Top 10 Botnet Threat Report - 2010*. Damballa Inc., 2010. Disponível em: <https://www.damballa.com/downloads/r_pubs/Damballa_2010_Top_10_Botnets_Report.pdf>.

DLLS. Microsoft, 2010. Disponível em: <http://msdn.microsoft.com/en-us/library/1ez7dh12(v=vs.71).aspx>.

FERRIE, P. *Anti-Unpacker Tricks*. [S.l.]: Microsoft Corporation, 2010.

GOSTEV, A. *Monthly Malware Statistics: February 2012*. SecuritList.com, 2012. Disponível em: <http://www.securelist.com/en/analysis/204792223/Monthly_Malware_Statistics_February_2012>.

HARBOUR, N. Advanced software armoring and polymorphic kung-fu. 2011.

HONEYCUTT, J. *Microsoft Windows Registry Guide*. [S.l.]: Microsoft Press, 2005. ISBN 978-0735622180.

INFORMIT. *Data Protection and Recovery Techniques Part 4: Using Norton Disk Editor*. Pearson Education, 2002. Disponível em: <http://www.informit.com/articles/article.aspx?p=339042>.

KASPERSKY. *Computer Threats FAQ*. Kaspersky, 2012. Disponível em: <http://www.kaspersky.com/threats_faq#ddos>.

KASPERSKY. *Computer Threats FAQ*. Kaspersky, 2012. Disponível em: <http://www.kaspersky.com/threats_faq#botnet>.

KAUSHAL, K.; SWADAS, P.; PRAJAPATI, N. Metamorphic malware detection using statistical analysis. *IJSCE*, v. 2, p. 49–53, 2012.

KRUEGEL, C. et al. Polymorphic worm detection using structural information of executables. 2005.

KRUEGEL, C. et al. Static disassembly of obfuscated binaries. 2004.

MATROSOV, A.; RODIONOV, E. Account of an investigation into a cybercrime group. 2012.

MICROSOFT. *Overview of the Windows API*. Microsoft, 2010. Disponível em: <http://msdn.microsoft.com/pt-br/library/aa383723.aspx>.

MICROSOFT. *Dynamic-Link Libraries*. Microsoft, 2012. Disponível em: <http://msdn.microsoft.com/en-us/library/windows/desktop/ms682589(v=vs.85).aspx>.

MICROSOFT. *PEB structure*. Microsoft, 2012. Disponível em: <http://msdn.microsoft.com/pt-BR/library/aa813706(v=vs.85).aspx>.

MICROSOFT. *PEB_LDR_DATA structure*. Microsoft, 2012. Disponível em: <http://msdn.microsoft.com/pt-BR/library/aa813708(v=vs.85).aspx>.

MICROSOFT. *RAM, virtual memory, pagefile, and memory management in Windows*. Microsoft, 2012. Disponível em: <http://support.microsoft.com/kb/2160852>.

MICROSOFT. *RTL_USER_PROCESS_PARAMETERS structure*. Microsoft, 2012. Disponível em: <http://msdn.microsoft.com/library/aa813741(VS.85).aspx>.

NAIR, V. P. et al. Medusa: Metamorphic malware dynamic analysis using signature from api. 2010.

NAMESTNIKOV, Y. *IT Threat Evolution: Q2 2012*. SecuritList, 2012. Disponível em: <http://www.securelist.com/en/analysis/204792239/IT_Threat_Evolution_Q2_2012>.

NTFS. *NTFS vs FAT vs exFAT*. 2011. Disponível em: <www.ntfs.com/ntfs_vs_fat.htm>.

NTFS. *NTFS File Attributes*. NTFS.com, 2012. Disponível em: <http://www.ntfs.com/ntfs-files-types.htm>.

RUSSINOVICH, M. E.; SOLOMON, D. A. *Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*. 4th. ed. [S.l.]: Microsoft Press, 2005. 3–5 p.

RUSSINOVICH, M. E.; SOLOMON, D. A.; IONESCU, A. *Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7*. 6th. ed. [S.l.]: Microsoft Press, 2012. 2–4 p.

SCHUSTER, A. *Memory Analysis: "Reconstructing a Binary (1)"*. 2006. Disponível em: <http://computer.forensikblog.de/en/2006/04/reconstructing-a-binary-1.html>.

SCHUSTER, A. *Memory Analysis: "_EPROCESS version 6.0.6000.16386"*. 2007. Disponível em: <http://computer.forensikblog.de/en/2007/01/eprocess-6-0-6000-16386.html>.

SHARIF, M. et al. Eureka: A framework for enabling static malware analysis. 2008.

SOCIALENGINEER. *Social Engineering Yourself A BotNet*. SocialEngineer.org, 2012. Disponível em: <http://www.social-engineer.org/social-engineering/social-engineering-yourself-a-botnet/>.

SOLOMON, D. A.; RUSSINOVICH, M. E. *Inside Microsoft Windows 2000*. [S.l.]: Microsoft Press, 2000. 277–317 p. ISBN 0-7356-1021-5.

STEVENS, K. *The Underground Economy of the Pay-Per-Install (PPI) Business*. SecureWorks Counter Threat Unit, 2010. Disponível em: <http://www.blackhat.com/presentations/bh-dc-10/Stevens_Kevin/BlackHat-DC-2010-Stevens-Underground-wp.pdf>.

STEWART, J. *Top Spam Botnets Exposed*. DELL - Secure Works, 2008. Disponível em: <http://www.secureworks.com/research/threats/topbotnets/>.

TIINSIDE. *TI Inside Online*. TIInside, 2011. Disponível em: <http://www.tiinside.com.br/30/11/2011/investimentos-em-seguranca-da-informacao-devem-crescer-em-2012-indica-pesquisa/sg/252033/news.aspx>.

WANG, C. et al. Using api sequence and bayes algorithm to detect suspicious behavior. 2009.

WINDOWS registry information for advanced users. Microsoft Support, 2008. Disponível em: <http://support.microsoft.com/kb/256986>.

YASON, M. V. *The Art of Unpacking*. [S.l.]: IBM internet Security System, 2007.

YOU, I.; YIM, K. Malware obfuscation techniques: A brief survey. 2010.

YURY, N. *Kaspersky Security Bulletin. Statistics 2011*. SecuritList.com, 2012. Disponível em: <http://www.securelist.com/en/analysis/204792216/Kaspersky_Security_Bulletin_Statistics_2011>.