Capítulo 3

Técnicas para Análise Dinâmica de *Malware*

Dario Simões Fernandes Filho[1], Vitor Monte Afonso[1], Victor Furuse Martins[1], André Ricardo Abed Grégio[1,2], Paulo Lício de Geus[1], Mario Jino[1], Rafael Duarte Coelho dos Santos[3]
[1]Universidade Estadual de Campinas (Unicamp)
[2]Centro de Tecnologia da Informação Renato Archer (CTI/MCT)
[3]Instituto Nacional de Pesquisas Espaciais (INPE/MCT)

Abstract

The threat posed by malware to systems security led to the development of analysis mechanisms that operate in a dynamic and controlled manner. These mechanisms (dynamic analysis systems) use a variety of techniques to obtain the behavior from malware samples' execution. The complexity of those techniques varies from monitoring events on user-level interfaces, to Web malicious code desobfuscation, to hooking operating system (OS) kernel structures. In this book chapter, we present the main techniques that are used to perform malware dynamic analysis, either at the operating system level or at the Web applications level. These techniques are used in the main publicly available analysis systems. Also, we show some tools used to capture information about the execution of malware samples and how to build a simple system to analyze malware using open-source and free tools. Finally, we describe in details a case study about the analysis of a malware sample that starts its attack from the browser and then compromises the OS.

Resumo

A ameaça dos códigos e programas maliciosos à segurança dos sistemas computacionais fez com que surgissem muitos sistemas cujo propósito é analisar, de maneira dinâmica e controlada, tais programas. Estes sistemas se utilizam de diversas técnicas para obter o comportamento apresentado por amostras de malware durante sua execução. A complexidade destas técnicas varia desde a monitoração de eventos através de interfaces no nível de privilégio dos usuários, passando pela desofuscação de programas maliciosos em linguagens típicas da Web, até a inserção de código em estruturas do kernel do

sistema operacional. Neste capítulo, visa-se apresentar as principais técnicas utilizadas para efetuar a análise dinâmica de malware - sejam estes do nível do sistema operacional ou da Web - e as quais estão presentes nos principais sistemas de análise disponíveis publicamente. Além disso, são mencionadas algumas ferramentas utilizadas na captura de informações da execução dos programas maliciosos. Mostra-se, também, como construir um sistema simples de análise dinâmica de malware utilizando ferramentas gratuitas ou de código aberto. Para finalizar o capítulo, os leitores terão a oportunidade de acompanhar um estudo de caso completo da análise de um exemplar de malware, do ataque a partir da Web até o comprometimento do sistema operacional.

3.1. Código Malicioso

Ataques realizados por meio de *malware* tomaram uma dimensão tão grande que atividades simples como a navegação na Web [Chen et. al. 2011], [Cova et. al. 2010], a participação em redes sociais digitais [Stringhini et. al. 2010], [Yang et. al. 2011] e o uso de celulares[Becher et. al. 2011], [Egele et. al. 2011] tornaram-se perigosas. Para se ter uma idéia do cenário nacional, quase a totalidade dos 142.844 incidentes reportados ao CERT.br[1] entre janeiro e dezembro de 2010 referem-se a ataques envolvendo código malicioso. Isto pode ser observado na Figura 3.1.

Figura 3.1. Incidentes reportados ao CERT.br em 2010. Fonte: http://www.cert.br/stats/.

Um estudo recente publicado na *IBM Systems Magazine* [IBM 2011] mostra que um dos maiores culpados pelo alto custo financeiro causado por um ataque é relacionado à atividade de *malware*. Na Figura 3.2 é mostrado o resultado do levantamento do custo de invasões em diversos países no ano de 2009.

No decorrer do tempo, em intervalos determinados, pode-se notar maior incidência de grupos específicos de *malware*. Por exemplo, nos últimos cinco anos tem havido uma proliferação grande de *botnets*[2], gerando tentativas (muitas vezes frustradas) do fechamento destas por parte de autoridades e pesquisadores de segurança [Stone-Gross et. al. 2011], [Zhang et. al. 2011], [Shin et. al. 2011], [Cho et. al. 2010].

[1]Centro de Estudos, Resposta e Tratamento de Incidentes de Segurança no Brasil. (Link: `http://www.cert.br`).

[2]redes de máquinas comprometidas controladas remotamente por comandos dados por um atacante.

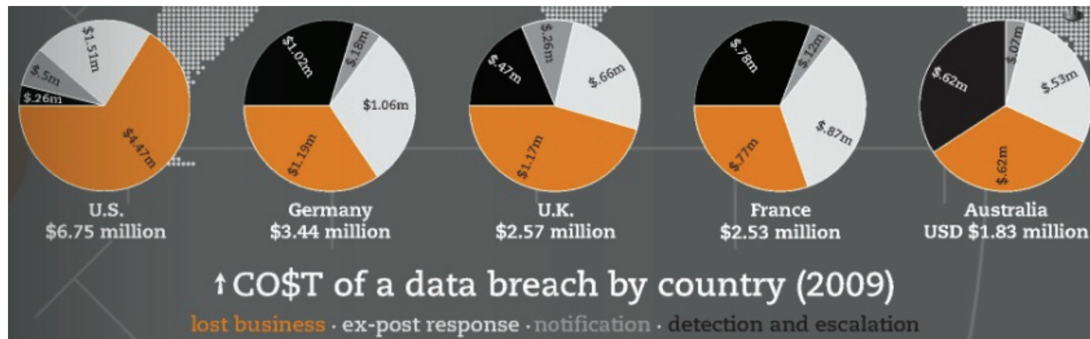

Figura 3.2. Custo de uma invasão por país em 2009. Fonte: [IBM 2011].

A incidência pontual de certas classes de *malware* faz com que estes sejam frequentemente responsáveis por sérios incidentes de proporções globais, tais como os recentes resultados do alastramento dos *malware* StuxNet [Falliere et. al. 2010], Zeus [Binsalleeh et. al. 2010] e Conficker [Shin and Gu 2010], ou casos que ganharam notoriedade no passado como, por exemplo, ILoveYou (2000), Nimda (2001), Code Red (2001), Slammer (2003), Blaster/Sasser (2004).

Embora tais grupos, ou famílias de *malware*, tendam a explorar as vulnerabilidades do momento e do contexto no qual estão inseridos, é comum a revisitação de antigos ataques, o compartilhamento de funções com finalidade maliciosa—varreduras, conexões com servidores, métodos de exploração—ou mesmo a mutação de pequenos trechos do código, a qual pode levar à redução da detectabilidade das famílias cujas vacinas já são conhecidas e, em paralelo, manter as funcionalidades e o comportamento malicioso apresentados nos ancestrais.

No âmbito da defesa, a mera identificação de um arquivo executável como sendo um *malware* conhecido (já coletado, analisado e talvez combatido) permite a tomada de contra-medidas de maneira rápida e eficiente. Isto facilita a contenção de danos, minimiza prejuízos e reduz a possibilidade de infecção em redes e sistemas ainda intactos por meio de regras de bloqueio ou aplicação de *patches* de segurança.

Com isso, é necessário haver meios de se identificar *malware* para que ações defensivas possam ser coordenadas, ao mesmo tempo em que sejam obtidos conhecimentos sobre o comportamento de um certo *malware* e sobre a possível extensão dos danos aos sistemas comprometidos por este código específico. A solução mais tradicional para identificação de *malware* ainda é o uso de antivírus. Porém, é cada vez mais comum a utilização de sistemas de análise dinâmica para traçar a execução de um programa e verificar por ações maliciosas. Adiante, será apresentada uma taxonomia sucinta de algumas classes importante de *malware*. Para motivar o leitor na identificação de comportamentos maliciosos, serão discutidos brevemente problemas relacionados ao emprego de mecanismos antivírus, da análise estática de *malware* e, por fim, da análise dinâmica, que é o tema deste texto.

3.1.1. Classes de código malicioso

Códigos maliciosos podem ser classificados de uma maneira geral de acordo com alguma faceta específica de seu comportamento. Embora atualmente seja difícil "enquadrar" um exemplar de *malware* em uma única classe, devido à evolução destes códigos e à facilidade de se adicionar novas funcionalidades, a taxonomia apresentada a seguir ainda é utilizada para se referir a certos tipos de *malware* e também nos identificadores atribuídos por mecanismos antivírus.

As classes e descrições abordadas a seguir baseiam-se nas definições de Peter Szor [Szor 2005].

Vírus. Segundo Fred Cohen, considerado o pai dos vírus de computador, "*um vírus é um programa que é capaz de infectar outros programas pela modificação destes de forma a incluir uma cópia possivelmente evoluída de si próprio*." Os vírus costumam infectar seções de um arquivo hospedeiro, propagando-se através da distribuição deste arquivo. Comumente, os vírus necessitam ser acionados e propagados por uma entidade externa (e.g., usuário).

Worm. Os *worms*, por sua vez, propagam-se pela rede e em geral não necessitam de ativação por parte do usuário. Uma outra característica é a independência de outro programa para disseminação e ataque. Alguns *worms* podem carregar dentro de si um outro tipo de *malware*, que é descarregado na vítima após o sucesso da propagação e do ataque.

Trojan. Cavalos-de-tróia são tipos comuns de *malware* cujo modo de infecção envolve despertar a curiosidade do usuário para que este o execute e comprometa o sistema. Este tipo de código malicioso também pode ser encontrado em versões modificadas de aplicações do sistema operacional, substituídas por indivíduos maliciosos. Estas versões apresentam as mesmas funcionalidades da aplicação íntegra, porém também contêm funcionalidades adicionais com a finalidade de ocultar as ações malignas.

Backdoor. Tradicionalmente, as *backdoors* permitem a um atacante a manutenção de uma máquina comprometida por abrirem portas que permitem a conexão remota. Um outro tipo de *backdoor* envolve erros na implementação de uma aplicação ou sistema que o tornam vulnerável e podem levar à execução de código arbitrário na máquina da vítima

Downloader. Um programa malicioso que conecta-se à rede para obter e instalar um conjunto de outros programas maliciosos ou ferramentas que levem ao domínio da máquina comprometida. Para evitar dispositivos de segurança instalados na vítima, é comum que *downloaders* venham anexos à mensagens de correio eletrônico e, a partir de sua execução, obtenham conteúdo malicioso de uma fonte externa. (e.g., Web site).

Dropper. Possui características similares as dos *downloaders*, com a diferença que um *dropper* é considerado um "instalador", uma vez que contém o código malicioso compilado dentro de si.

Rootkit. É um tipo especial de *malware*, pois consiste de um conjunto de ferramentas para possibilitar a operação em nível mais privilegiado. Seu objetivo é permanecer residindo no sistema comprometido sem ser detectado e pode conter *exploits*, *backdoors* e versões *trojan* de aplicações do sistema. Os *rootkits* modernos atacam o *kernel* do sistema

operacional, modificando-o para que executem as ações maliciosas de modo camuflado. Este tipo de *rootkit* pode inclusive interferir no funcionamento de mecanismos de segurança.

3.1.2. O problema dos antivírus

Um dos mecanismos de defesa contra *malware* mais populares ainda é o antivírus, que pode ser explicado basicamente como um programa que varre arquivos ou monitora ações pré-definidas em busca de indícios de atividades maliciosas. Em geral, os antivírus operam de duas formas para identificar código malicioso: correspondência de padrões em bancos de dados de assinaturas ou heurísticas comportamentais. Na detecção por assinatura, um arquivo executável é dividido em pequenas porções (*chunks*) de código, as quais são comparadas com a base de assinaturas do antivírus. Assim, se um ou mais *chunks* do arquivo analisado estão presentes na base de assinaturas, a identificação relacionada é atribuída ao referido arquivo. Já na detecção por heurística, um arquivo sob análise é executado virtualmente em um emulador minimalista e os indícios de comportamento suspeito são avaliados a fim de se verificar se a atividade realizada pelo programa pode ser considerada normal ou se um alerta deve ser emitido.

O grande problema dos antivírus é o surgimento frequente e crescente de variantes de *malware* previamente identificados, cujas ações modificadas visam evadir a detecção. Essas variantes precisam ser tratadas muitas vezes individualmente (e manualmente), no caso da confecção de uma assinatura para um antivírus. Além do mais, pode ser preciso modificar a heurística de detecção de toda uma classe para que esta seja capaz de identificar a variante, sempre levando-se em conta se a modificação não vai gerar mais falsos-positivos. Esses, por sua vez, são uma outra dificuldade no processo de criação de assinaturas/heurísticas de *malware*—identificar erroneamente uma aplicação inofensiva do usuário como sendo maliciosa e, consequentemente, removê-la, colocá-la em quarentena (bloqueio) ou restringir de algum modo a usabilidade de um sistema computacional não é um requisito desejável para os fabricantes de antivírus.

Deve-se considerar também que muitos exemplares de *malware* possuem mecanismos próprios de defesa cujas ações variam entre desabilitar as proteções existentes no sistema operacional alvo (*firewall*, antivírus, *plugins* de segurança), verificar se o exemplar está sob análise (o que pode causar uma modificação em seu comportamento em razão disto), e disfarçar-se de programas do sistema, inclusive de falsos antivírus. Portanto, evitar que o mecanismo antivírus seja desabilitado, identificar programas maliciosos rapidamente com uma taxa mínima de falsos-positivos sem interferir na usabilidade do sistema e, ainda, expandir a capacidade de detecção sem causar sobrecarga têm se tornado tarefas cada vez mais difíceis. Contribui a isso o cenário atual das variantes, popularização das redes sociais, proliferação de celulares e *tablets* com o mesmo poder e funcionalidade de computadores, e o estabelecimento da *cloud computing*.

Não bastassem os problemas supracitados, a comunidade de desenvolvedores de antivírus ainda não possui padronização para a classificação dos *malware* identificados por suas ferramentas. Isso faz com que o processo de resposta a incidentes de segurança envolvendo código malicioso seja prejudicado, tornando-o ineficiente em determinados casos. De modo ilustrativo, pode-se tomar como exemplo alguns sistemas que identificam

um programa como malicioso baseando-se apenas no *packer*[3] com o qual o arquivo foi cifrado, ou ainda, que fazem a identificação baseada em apenas uma faceta do binário, a qual pode não condizer com as atividades efetuadas na máquina[4]. Divergências de nomenclatura terminam por atrapalhar na classificação, pois a falta de um padrão faz com que cada fabricante de antivírus atribua o identificador que desejar. Assim, um dado *malware* pode ser identificado como 'Família_X.A' por um fabricante, 'Família_X.123' por outro e 'Família_Z.y' por um terceiro. Para ilustrar este problema de forma mais clara, na Tabela 3.1 estão os resultados obtidos pela indentificação de um exemplar de *malware* submetido ao VirusTotal[5].

Tabela 3.1. Resultado do VirusTotal para um exemplar de código malicioso.

Antivirus	*Identificação*
McAfee	**PWS-OnlineGames.bu**
K7AntiVirus	**Trojan**
NOD32	**a variant of Win32/PSW.OnLineGames.OAF**
F-Prot	**W32/Agent.L.gen!Eldorado**
Symantec	**Infostealer.Gampass**
Norman	**W32/Packed_Upack.H**
TrendMicro-HouseCall	**TSPY_ONLING.AG**
Avast	**Win32:OnLineGames-ECV [Trj]**
eSafe	**Win32.Looked.gen**
Kaspersky	**Trojan-GameThief.Win32.OnLineGames.akyb**
Sophos	**Mal/Behav-214**
Comodo	**TrojWare.Win32.Trojan.Inject.ĨI**
DrWeb	**Trojan.DownLoader.62898**
VIPRE	**BehavesLike.Win32.Malware.bsu (vs)**
AntiVir	**TR/Spy.Gen**
TrendMicro	**TSPY_ONLING.AG**
McAfee-GW-Edition	**Heuristic.BehavesLike.Win32.Packed.A**
eTrust-Vet	**Win32/Lemir!generic**
AhnLab-V3	**Win-Trojan/Xema.variant**
VBA32	**BScope.Trojan-PSW.SataGames.3**
PCTools	**Rootkit.Order**
Rising	**Trojan.PSW.Win32.GameOL.odt**
Ikarus	**Virus.Win32.Virut**
Panda	**Trj/Lineage.ISP**

Para a resposta ser efetiva, isto é, permitir a remoção do código malicioso e as atividades derivadas da infecção do sistema comprometido, é necessário, em muitos ca-

[3]No contexto deste texto, um *packer* é um programa utilizado para cifrar ou comprimir o conteúdo de um arquivo/programa malicioso para disfarçá-lo.

[4]Exemplares de *malware* infectados por outros *malware* podem fazer com que o antivírus emita um alerta devido a assinatura do infectante, e não do *malware* original infectado.

[5]VirusTotal é um serviço gratuito disponibilizado publicam ente na Internet em `http://www.virustotal.com`, para o qual podem ser submetidos programas que são analisados por grande parte das ferramentas antivírus atualmente disponíveis no mercado.

sos, uma intervenção manual, dado que nem todos os antivírus possuem o procedimento de desfazimento das ações efetuadas (nem há tal procedimento para todas as amostras de *malware* em atividade, devido ao já citado problema da existência massiva de variantes). Alguns casos mais epidêmicos acabam por ter aplicações ou rotinas automatizadas para remoção, as quais podem ser inócuas na presença de variantes. Entretanto, o sucesso na obtenção do procedimento de remoção (automático ou manual) correto é completamente dependente da identificação provida pelo mecanismo antivírus que, como visto anteriormente, pode ter sido feita de maneira errônea.

3.1.3. Análise estática e suas limitações

A análise de código malicioso visa o entendimento profundo do funcionamento de um malware—como atua no sistema operacional, que tipo de técnicas de ofuscação são utilizadas, quais fluxos de execução levam ao comportamento principal planejado, se há operações de rede, download de outros arquivos, captura de informações do usuário ou do sistema, entre outras atividades. Divide-se a análise de malware em análise estática e dinâmica, sendo que no primeiro caso tenta-se derivar o comportamento do malware extraindo características de seu código sem executá-lo, através de análise de strings, disassembling e engenharia reversa, por exemplo. Já na análise dinâmica, o malware é monitorado durante sua execução, por meio de emuladores, debuggers, ferramentas para monitoração de processos, registros e arquivos e tracers de chamadas de sistema. Dependendo da técnica ou ferramenta que se utiliza para fazer cada análise, a velocidade pode variar, mas, em geral, análises estáticas simples são mais rápidas do que as dinâmicas. Entretanto, se há a necessidade de engenharia reversa, se o malware possui muitos fluxos de execução ou se está comprimido com um packer de difícil descompressão, a análise dinâmica tradicional é muito mais rápida e eficaz na provisão de resultados acerca do comportamento do exemplar analisado. Em [Moser et al. 2007] é apresentada uma ferramenta que transforma um programa de forma a ofuscar seu fluxo de execução, disfarçar o acesso a variáveis e dificultar o controle dos valores guardados pelos registradores, mostrando que a análise estática de um malware ofuscado por essa ferramenta é um problema NP-difícil. Além disso, durante a análise estática não se sabe como o sistema vai reagir em resposta às operações do programa.

A análise estática pode ser utilizada para obter informações gerais sobre o programa e para identificar a existência de código malicioso. Dentre as técnicas utilizadas para a obtenção de informações gerais estão a geração de hashes criptográficos que identificam o arquivo de forma única, a identificação das funções importadas e exportadas, a identificação de código ofuscado e a obtenção de cadeias de caracteres que possam ser lidas por uma pessoa, como mensagens de erro, URLs e endereços de correio eletrônico. Para identificar código malicioso são usadas, de forma geral, duas abordagens: a verificação de padrões no arquivo binário e a análise do código assembly gerado a partir do código de máquina do malware. No caso da verificação de padrões, são geradas seqüências de bytes, chamadas de assinaturas, que identificam um trecho de código freqüentemente encontrado em programas maliciosos e verifica-se se o programa possui esta seqüência. Já no caso da investigação do código assembly, são empregadas técnicas de análise mais profundas que buscam padrões de comportamento malicioso. Em [Song et al. 2008] os autores transformam o código assembly em uma linguagem intermediária e, a partir

desta, extraem informações a respeito do fluxo de dados e fluxo de controle do programa. A maior dificuldade encontrada pela análise estática é o uso dos packers. Para combater a evolução destes, foram desenvolvidos diversos mecanismos [Yegneswaran et al. 2008] [Kang et al. 2007] [Martignoni et al. 2007] que visam obter o código não ofuscado do malware, permitindo que a análise estática seja efetuada.

3.1.4. Analisadores dinâmicos

Embora o *modus operandi* dos antivírus ainda seja majoritariamente a identificação com base em assinaturas, tem crescido o interesse por heurísticas, pois uma heurística bem construída pode resultar na substituição de dezenas de assinaturas. Para gerar uma heurística que identifique um exemplar de *malware* (ou uma classe), é necessário conhecer primeiro o seu comportamento, isto é, quais são as ações realizadas no sistema operacional alvo que denotam uma atividade anormal ou suspeita. Como opção aos emuladores limitados embutidos nos antivírus com o objetivo de realizar a identificação por heurísticas, os sistemas de análise dinâmica de *malware* foram sendo aprimorados e popularizados nos últimos anos. Tais sistemas lançam mão de uma variedade de técnicas para monitorar a execução de um *malware* de maneira controlada, utilizando desde a instrumentação de emuladores complexos até a interceptação de chamadas ao *kernel* do sistema operacional monitorado.

É comum as empresas fabricantes de antivírus possuirem seus próprios sistemas de análise dinâmica de *malware*, também chamados de *sandboxes*, mas há soluções disponíveis gratuita e publicamente via Internet, como *Anubis*[6], *ThreatExpert*[7] e *CWSandbox*[8]. No contexto deste trabalho, uma *sandbox* é um ambiente restrito e controlado que permite a execução de um código malicioso de forma a causar danos mínimos à sistemas externos por meio da combinação de filtragem e bloqueio de tráfego de rede e da execução temporizada do *malware*. Em geral, o *malware* é executado por quatro ou cinco minutos e, durante este tempo, são monitoradas as ações pertinentes tanto ao *malware* quanto aos processos derivados dele. Após o período de monitoração, um relatório de atividades é gerado para análise.

Entretanto, um dos grandes problemas da análise dinâmica é que a interpretação dos relatórios é deixada a cargo do usuário que submeteu o exemplar, ou seja, não se pode realmente dizer que o sistema fez uma análise, mas sim, uma monitoração da execução com registro das atividades efetuadas no período. Além disso, limitações das técnicas comumente utilizadas para interceptar as chamadas de sistema ou instrumentar o ambiente da *sandbox* podem levar à evasão da análise por exemplares de *malware* modernos. Isso, agravado pelo fato de alguns exemplares de *malware* terem por característica a mutação de seus comportamentos durante execuções distintas, faz com que o relatório gerado por um sistema possa ser diferente do relatório gerado por outro, no que diz respeito às ações efetuadas pelo *malware* sob análise.

Ainda assim, a análise dinâmica é um importante instrumento para prover informações úteis a um usuário ou analista de segurança, permitindo tomadas de decisão com

[6] `http://anubis.iseclab.org`
[7] `http://www.threatexpert.com`
[8] `http://mwanalysis.org`

base no padrão de ações nocivas executadas pelo exemplar de *malware* analisado.

3.1.5. Organização do texto e considerações

Este texto visa cobrir as técnicas mais utilizadas em análise dinâmica de *malware*, sejam eles voltados aos ataques via Web ou no nível do sistema operacional e suas aplicações. Na Seção 3.2 são apresentadas algumas destas técnicas, enquanto que as ferramentas que as utilizam são mostradas na Seção 3.3. É esperado que o leitor compreenda os pontos fortes e as limitações de cada técnica, bem como o funcionamento destas de forma a poder escolher aquela que se adapta às suas necessidades de análise. Um conjunto de ferramentas para a extração de informações da execução de um código malicioso é mostrado na Seção 3.4. As etapas da análise serão ilustradas por meio de exemplos práticos, permitindo o seu acompanhamento detalhado em um estudo de caso apresentado na Seção 3.6.

3.2. Técnicas de Análise

A análise dinâmica de *malware*, normalmente realizada para identificar as ações desenvolvidas pelo *malware* no sistema, pode ser implementada através de diversas técnicas, cada uma com suas vantagens e desvantagens. Neste capítulo serão apresentadas as principais técnicas utilizadas por sistemas de análise importantes ou por ferramentas que auxiliam na análise dinâmica. Dentre elas, podemos citar *Virtual Machine Introspection* e *Hooking* como as principais. Cada subseção irá tratar cada uma delas de forma detalhada, de forma que ao final deste capitulo o leitor será capaz de compreender cada uma delas, podendo escolher dentre elas a que mais se adequar para a sua análise.

3.2.1. *Virtual Machine Introspection*

Virtual Machine Introspection (VMI) é uma técnica onde se cria uma camada entre o sistema de análise (*guest*) e o ambiente de processamento (*host*), de forma que todas as ações que ocorrem dentro do sistema *guest* não são propagadas para o *host*. Seu uso possibilita a captura das ações que estão sendo executadas dentro do ambiente de análise, sem que haja qualquer interferência dentro do ambiente onde está sendo executado, possibilitando assim que um *malware* seja análise sem qualquer tipo de modificação no sistema *guest*. Como não há modificação no sistema *guest*, a análise fica transparente para o *malware*, impossibilitando qualquer tentativa de identificação do componente de captura. A ilustração apresentada na figura 3.3 visa clarificar o funcionamento da técnica de *Virtual Machine Introspection*, que será tratada mais detalhadamente nas seções seguintes. É possível verificar a diferenciação entre os sistemas *guest*, que funcionam dentro do ambiente emulado e o sistema *host*, onde está executando a aplicação responsável pela implementação da camada entre o ambiente *guest* e o *host*.

Programas para emulação e virtualização, tais como Qemu [Bellard 2005], VMWare [VMware 2011] e VirtualBox [Virtualbox 2011] possibilitam a aplicação desta técnica, dado que estes implementam a camada intermediária de maneira nativa, fazendo assim uma distinção entre o ambiente real (ou sistema *host*) e o emulado/virtualizado (ou o *guest*). A seguir serão explicados os conceitos de emulação e como a técnica de VMI pode ser aplicado à ela.

Emulação é um termo utilizado na área de computação para descrever o modo de

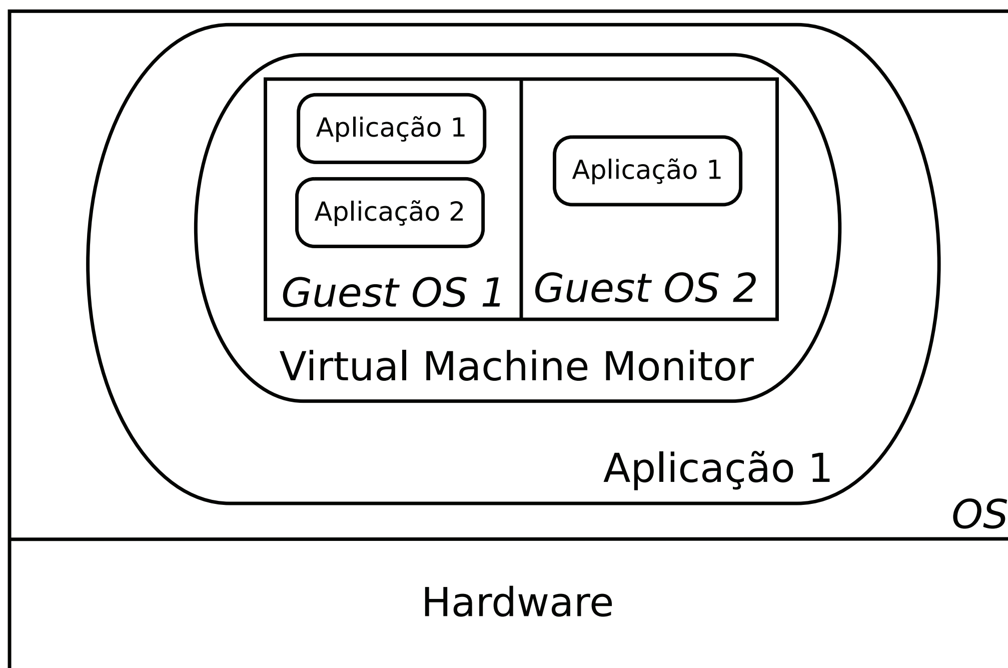

Figura 3.3. Ilustração do uso da técnica de *Virtual Machine Introspection*.

operação de um *software* desenvolvido para simular um determinado *hardware* [Martignoni et. al. 2009], como por exemplo um processador específico diferente do que está executando o emulador. Na análise de *malware*, este tipo de *software* é utilizado para simular uma máquina, para que seja possível a instalação do sistema operacional que será utilizado no processo de análise. Além disso, o emulador pode ser utilizado para separar o ambiente de análise, fazendo com que as ações efetuadas por um exemplar de *malware* durante sua execução não contaminem o ambiente real, dado que as modificações executadas pelo *malware* só irão ocorrer no ambiente de análise *host*. Um emulador muito utilizado para este fim é o Qemu [Bellard 2005]—uma ferramenta de código aberto e de fácil utilização com o qual é possível emular o processador, o disco rígido e demais dispositivos do sistema de forma que seja possível instalar vários tipos de sistemas operacionais.

Virtualização, de modo similar à emulação, é utilizada para simular uma máquina, tornando possível a instalação e execução de vários sistemas operacionais em paralelo com o mesmo *hardware*. Porém, na virtualização as instruções são executadas no *hardware* real da máquina, ao contrário do que ocorre na emulação, na qual as instruções são executadas em um processador emulado. Isto torna a virtualização mais rápida em relação à emulação, pois as instruções do *guest* ficam sob responsabilidade do *hardware* do *host*. A limitação da virtualização é que ela possibilita somente a instalação de sistemas cuja arquitetura seja a mesma do *host*. Quando se utiliza um virtualizador para análise de *malware*, as funcionalidades são parecidas com as do emulador: há o isolamento entre os ambientes dos sistemas *host* e *guest*. Nesse caso, o programa responsável pela virtualização, acrescenta uma camada adicional entre o ambiente real e o de análise chamada de *Virtual Machine Monitor*. Esta camada realiza a abstração do *hardware* real para as

máquinas virtuais, executando suas ações de forma que sejam percebidas somente dentro do ambiente virtual [Rosenblum 2004].

Tanto a virtualização quanto a emulação fazem uma distinção entre o ambiente onde o *malware* é executado e o real. Para simplificar a forma de mencionar tais ambientes, a partir de agora serão utilizados os termos *guest* e *host*, sendo que o primeiro identifica o sistema operacional virtualizado ou emulado utilizado na análise dinâmica e o último identifica o sistema base que executa o emulador ou virtualizador.

O programa de virtualização/emulação responde ao sistema *guest* da mesma forma que os dispositivos físicos (processador, disco rígido, placas de rede e vídeo etc) responderiam, sem entretanto comprometer o *host* no qual ele está sendo executado. Essa transparência faz com que o *host* tenha total controle sobre o *guest*, podendo inclusive observar em tempo real o estado dos diversos recursos da máquina onde o *malware* está sendo executado, como memória e CPU, por exemplo.

Desta forma, torna-se trivial a obtenção de informações a respeito da execução do *malware* de maneira externa ao *guest*, bastando que se modifique o *software* responsável por executar a emulação/virtualização para que este realize a captura dos dados. A modificação de um programa de virtualização ou emulação com o objetivo de se obter informações internas ao *guest* a partir do sistema *host* é chamada de VMI e, com a utilização desta técnica é possível alcançar um nível de privilégio adicional na camada de abstração intermediária entre o *host* e o *guest*. Uma das características mais interessantes da VMI é que esta torna possível a análise de *malware* cuja execução ocorre no nível do *kernel*, tais como os *rootkits*. As técnicas comumente utilizadas por *rootkits* para esconder ou alterar estruturas internas do sistema operacional atacados [Hoglund and Butler 2005] podem inviabilizar sua detecção e monitoração por mecanismos de segurança ou outros métodos de análise, como por exemplo, *hooking* (Seção 3.2.2).

Para ilustrar a técnica de VMI, um tipo de informação que pode ser capturada do sistema *guest* são as *syscalls* que o *malware* executou durante a análise. Um método muito utilizado para identificar a ocorrência de uma *syscall* baseia-se na leitura do valor contido no registrador `SYSENTER_EIP_MSR` do processador. Este registrador é utilizado quando ocorre uma instrução do tipo "`SYSENTER`", que indica que uma chamada de sistema deve ser feita. Quando a chamada é efetuada, o sistema realiza a troca de contexto entre o espaço de usuário e o espaço de *kernel*, permitindo finalmente a execução da *syscall*.

Uma forma de identificar qual *syscall* está sendo invocada é através da leitura do valor contido no registrador `EAX`. No momento em que a instrução `SYSTENTER` for executada, o registrador EAX armazena um valor utilizado para se encontrar o endereço da *syscall* que se quer realizar. Esse valor corresponde ao índice de uma tabela que contém os endereços de todas as *syscalls* possíveis no sistema operacional. Em sistemas *Windows*, esta tabela corresponde a uma estrutura que atende pelo nome de *System Service Dispatch Table*. Os parâmetros utilizados para compor a *syscall* podem ser obtidos através de verificações nos registradores do processador e na memória do sistema, no momento em que a chamada estiver sendo executada.

Um sistema de análise dinâmica bem conhecido e disponível publicamente para

utilização através da Internet é, *Anubis* [Anubis 2011], o qual utiliza a técnica de VMI para monitorar as ações de um exemplar de *malware* durante sua execução em um sistema operacional *Windows XP*. A fim de aplicar VMI, *Anubis* foi implementado sobre o emulador *Qemu*, modificado para efetuar a captura de informações de maneira externa ao *guest*. Mais detalhes da implementação de *Anubis* podem ser encontrados em [Bayer et. al. 2006].

A principal desvantagem da VMI é que um *malware* pode detectar que está sendo executado em um ambiente emulado/virtual, evitando a análise como um todo ou apresentando um comportamento alternativo ao malicioso. No caso dos emuladores, a detecção pode ser feita de forma muito simples, por exemplo, através da realização de uma instrução no processador que causa um comportamento específico. Um dos modos utilizados para realizar tal verificação é por meio de *bugs* conhecidos em processadores de determinadas arquiteturas que fazem com que certas instruções não se comportem como esperado. Se esta instrução for executada no emulador e este não estiver preparado para apresentar o mesmo comportamento de um processador real, o *malware* irá perceber essa diferença, podendo parar ou modificar a sua execução [Raffetseder et. al. 2007]. A detecção de ambiente virtualizado também é simples, com apenas uma instrução *assembly* que, mesmo executada em um nível de baixo privilégio, retorna informações internas sobre o sistema operacional presente no *guest*. Tais informações identificam o ambiente virtualizado com base nas diferenças entre estes e sistemas reais [Quist and Smith 2006].

Para contornar as técnicas de anti-análise, existem meios de detectar que um *malware* verifica se está em ambiente emulado, ou mesmo de modificar alguns valores presentes no ambiente virtual para tentar disfarçá-lo [Kang et. al. 2009],[Liston and Skoudis 2006]. Entretanto, o uso destas técnicas muitas vezes é insuficiente e o *malware* ainda pode detectar que está sendo executado em ambiente emulado/virtual.

Um outro sistema utilizado para traçar o comportamento de um *malware* e que se baseia em VMI é *Ether* [Dinaburg et. al. 2008]. Este sistema utiliza VMI para obter as ações realizadas no *guest*, porém, ao contrário de *Anubis*, *Ether* se utiliza de virtualização direta do *hardware*, o que o torna imune às técnicas de anti-análise que verificam se o *hardware* é real ou emulado. A implementação da VMI é feita em uma versão modificada do *Xen hypervisor*, um *software* de virtualização. Uma vantagem de *Ether* sobre *Anubis* diz respeito à análise de exemplares de *malware* com *packers* que apresentam mau funcionamento em emuladores. Diferentemente de *Anubis*, *Ether* consegue analisar este tipo de *malware* sem qualquer problema em sua execução, dado que o *Xen* utiliza o *hardware* nativo para executar as operações do processador. Entretanto, *Ether* apresenta problemas de desempenho para obter o traço composto pelas *syscalls* que o *malware* realizou. Isso ocorre porque cada chamada de sistema executada pelo programa gera uma *page fault*, a qual é tratada pelo componente de *Ether* responsável pela obtenção do referido traço. Além disso, apesar de ser dito em sua documentação que não é possível detectar sua presença, existem meios de verificá-la, como os descritos em [Pék et. al. 2011]. Nesta referência, são apontados alguns possíveis modos de detectar a presença de *Ether*, como por exemplo através de uma modificação feita pelo sistema de análise que desabilita o bit `TSC` (*Time-Stamp Counter*). Este bit é retornado quando se executa a instrução `CPUID` e serve para indicar quando a instrução *RDTSC* é suportada. Portanto, para detectar a execução em *Ether*, basta que se execute a instrução `CPUID` e se observe o valor retornado

no bit `TSC`.

Além dos problemas apresentados com o uso da VMI para captura de informações, há uma outra limitação que diz respeito ao desempenho. Como o componente que obtém as informações fica na camada da VMI, que faz o interfaceamento entre o *guest* e o *host*, os dados capturados são de nível mais baixo, isto é, valores encontrados em registradores da CPU ou endereços de memória. Porém, a análise do comportamento do *malware*, isto é, as modificações feitas no sistema da vítima, requer a obtenção de valores de mais alto nível, como nomes de arquivos criados, registros modificados e processos inicializados. Assim, para acessar tal conteúdo precisa-se interpretar, em tempo de execução, os dados contidos na memória e no processador durante a monitoração do *malware*, o que na maioria das vezes não é uma tarefa fácil e causa uma sobrecarga no processo de análise.

3.2.2. *Hooking*

A técnica de *hooking* pode ser definida como um meio de se alterar as requisições e respostas resultantes das interações realizadas em um sistema operacional ou por suas aplicações, através da interceptação das funções ou eventos utilizados [Holy Father 2004]. Pode-se categorizar *hooking* como sendo de modo de usuário (*userland hooking*) ou de modo de *kernel* (*kernel hooking*). O que difere estes dois tipos é a extensão da modificação que pode ser feita no sistema e, consequentemente, nas aplicações. *Malware* geralmente utilizam-se de técnicas de *hooking* para capturar ou modificar informações que estejam transitando em uma aplicação ou no sistema operacional. Através disto é possível a ocultação de suas atividades, dificultando assim a sua identificação. Alguns *rootkits* empregam *hooking* para tornar sua presença indetectável ao sistema [Hoglund and Butler 2005]. Na figura 3.4 é apresentada uma ilustração da aplicação da técnica de *SSDT hooking*, que será explicada com mais detalhes nas seções seguintes, em um sistema. Pode-se perceber como era o fluxo de execução antes do *hooking* e depois dele.

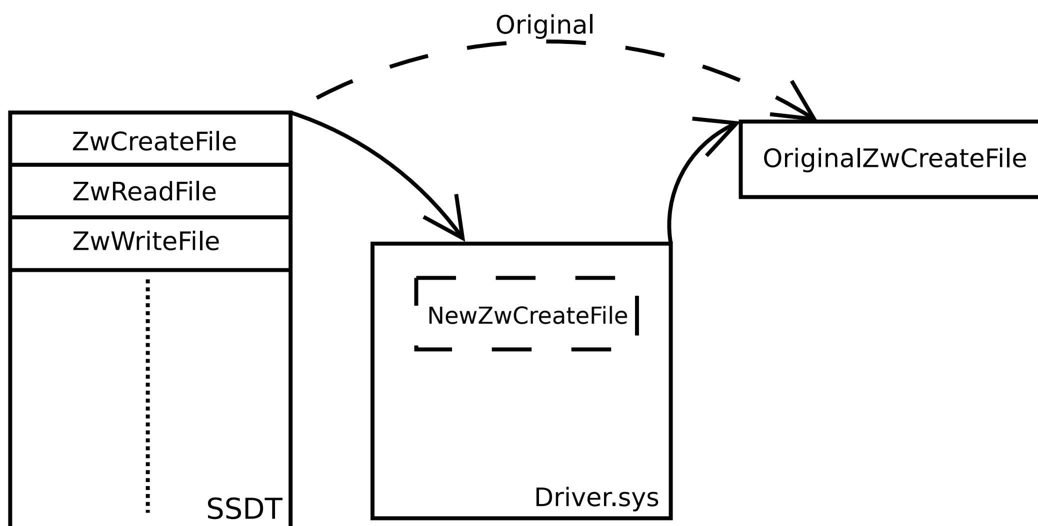

Figura 3.4. Ilustração da aplicação da técnica de *SSDT Hooking*.

Nas seções a seguir, serão detalhadas as diferenças existentes entre *hooking* de nível de usuário e de *kernel*. Serão citados também exemplos de cada uma das abordagens.

3.2.2.1. *Userland Hooking*

Userland hooking ou *hooking* de nível de usuário é uma técnica de interceptação que pode afetar somente programas que executam em nível de usuário, não podendo interferir em qualquer aplicação que opere em um nível mais privilegiado. Mesmo com esta limitação, tal técnica é bastante utilizada por *malware*, dado que sua implementação é mais simples. Embora sua utilização seja frequente, *userland hooking* pode ser facilmente detectado, o que pode levar o programa a desfazer o *hooking* ou não executar a função modificada. Como esse tipo de *hooking* é feito normalmente sobre APIs[9] disponibilizadas pelo sistema operacional, para um programa detectar se a interceptação está sendo feita ou não basta verificar se os endereços das APIs utilizadas por ele estão modificados ou se o endereço é uma instrução de pulo incondicional (JMP), que é uma prática comumente utilizada em *hooking*. Além deste tipo de detecção, existem outras formas que podem ser utilizadas para evitar um possível *hooking*. Uma delas, muito eficaz, é a utilização de funções nativas do sistema operacional, ao invés das APIs fornecidas por ele. Entretanto, empregar este processo requer um maior cuidado na implementação do programa, pois este deverá fornecer um número maior de informações quando for executar cada função, tarefa esta que antes ficava a cargo do sistema operacional. A fim de exemplificar técnicas de *userland hooking* para modificação de APIs, apresenta-se a seguir *IAT hooking*, *Detours* e *inline hooking*.

IAT Hooking

Import Address Table hooking, ou (*IAT hooking*), é uma técnica aplicada para interceptar as funções utilizadas por um determinado programa, antes que este esteja em execução. Para isso, a técnica deve ser empregada sem que o *malware* tenha comprometido o sistema. A *Import Address Table* é uma estrutura presente no cabeçalho de arquivos do tipo PE32 (arquivos executáveis do sistema *Windows*), e é responsável por indicar os endereços das rotinas externas utilizadas por um programa [Microsoft 2008]. Esses endereços são definidos no processo de carregamento do programa que antecede a sua execução, quando este está sendo inicializado pelo sistema operacional. Tais endereços são fornecidos por DLLs (*dynamic-link libraries*), pacotes binários que contêm funções e variáveis as quais podem ser utilizadas por outros programas [Russinovich and Solomon 2004]. Quando o programa está na fase de inicialização, o sistema operacional se encarrega de verificar quais DLLs e métodos externos são utilizadas por ele. De posse destas informações, o sistema operacional pode preencher a IAT com os endereços referentes aos métodos usados, de forma que durante a execução o programa carregado consiga invocar corretamente as funções [Holy Father 2004].

Para realizar este tipo de *hooking* é necessário modificar a tabela IAT do programa que se deseja monitorar, de forma que os endereços contidos nela sejam de funções que se tem controle. Portanto, para cada função modificada é necessária uma nova função, a qual poderá modificar ou simplesmente monitorar os dados passados pelo programa sob análise para a função original. Além disso, cada função deve invocar a função original para que as ações produzidas por um programa tenham seu efeito consumado no sistema, prosseguindo assim com a execução normal do programa monitorado. Um problema evi-

[9] Application Programming Interfaces

dente desta abordagem é a necessidade de modificações no programa sob análise, a fim de que seja possível instalar os *hookings* nas *APISs* monitoradas. A detecção deste tipo de ação pode ser feita com um simples teste de integridade no código do programa monitorado. Se isto ocorrer, um *malware* pode identificar que está sendo monitorado, o que pode fazê-lo tomar medidas que inviabilizem sua análise ou que os resultados retornados por esta não correspondam ao fluxo de execução malicioso pretendido originalmente. Outro problema com a abordagem que pode inutilizar a captura ocorre caso o *malware* carregue a DLL que irá utilizar durante a sua execução. Para isso, basta que ele use APIs disponibilizadas pelo *Windows*, as quais possibilitam que a DLL seja utilizada, mesmo sem ser previamente inicializada com o *malware*. Esta prática é bem simples de ser utilizada e desabilita por completo o *IAT hooking*, já que as APIs utilizadas assim pelo *malware* não serão afetadas.

Detours

Uma outra forma de interceptar *APIs* do *Windows* é através do uso de *Detours*, uma biblioteca provida pela própria *Microsoft* para interceptar funções do *Windows* em arquiteturas x86 [Hunt and Brubacher 1999]. Através de sua utilização, é possível realizar modificações no início da função que se deseja interceptar de maneira dinâmica, durante a execução do programa. Esta técnica é implementada através da inserção de uma instrução *assembly* de pulo incondicional (`JMP`) no início da função. Assim, quando tal função for invocada, o *JMP* será executado e irá direcionar o fluxo de execução para uma região sobre a qual se tem controle. Isto possibilita a captura dos dados que estão passando pela função, bem como os valores retornados com sua execução. Sua implementação pode ser feita de maneira simples, visto que o próprio sistema operacional provê suporte para isso, através de um programa que executa em nível de usuário.

Um sistema *open source* para análise dinâmica de *malware* disponibilizado recentemente, chamado *Cuckoobox* [Cukoo Sandbox 2011], aplica a técnica de *Detours* para obter as informações das APIs utilizadas pelo *malware* durante sua execução. Embora o sistema necessite de máquinas virtuais para realizar uma análise, o mecanismo de captura é inserido no *guest* e passa as informações obtidas via protocolo de comunicação para o *host*, diferentemente de *Anubis*, que requer uma modificação no *software* de emulação para capturar as informações de execução do *malware*. Isto torna a implementação da técnica mais simples, porém, seu custo é que pode-se facilmente verificar o uso de *Detours* através de uma checagem no início da função que se quer realizar: se a instrução inicial for um *JMP*, há a presença de um *Detour*.

Inline Hooking

Inline Hooking é uma outra técnica que pode ser utilizada para redirecionar o fluxo de execução normal de um programa para uma região que se tenha total controle. Em geral, esta técnica é utilizada em *malware* para alterar as APIs do sistema operacional, de forma que as respostas produzidas sejam capturadas ou modificadas. Cabe ressaltar que *inline hooking* pode ser implementada tanto no nível do usuário como no do *kernel*, entretanto, sua forma mais comum aparece em nível de usuário devido à simplicidade da implementação. Seu funcionamento é bem parecido com o do *Detours*, mas em vez de trocar somente a primeira instrução *assembly* da função que se deseja interceptar, é possível alterar uma porção maior de código. Como a parte que desvia a execução do

programa para a região de que se tem controle não fica no início do código, a detecção do *hooking* é mais complicada, visto que será necessário inspecionar uma área maior do código que se quer executar em busca de algum desvio de execução.

O sistema de análise de *malware CWSandBox* [CWSandbox 2011] utiliza a técnica de *inline hooking* para capturar as informações resultantes da execução de um *malware* em um sistema *Windows XP*. O *inline hooking* feito por ele é similar à técnica do *Windows Detour*, onde o começo da função que se deseja interceptar é substituído por um *JMP* [Willems et. al. 2007].

3.2.2.2. *Kernel Hooking*

O *kernel hooking*, ou *hooking* em nível de *kernel*, executa em um nível mais privilegiado, utilizando técnicas mais complexas que não são trivialmente detectadas por *malware*. Isto atribui uma vantagem sobre o *userland hooking*, pois torna a sua detecção mais difícil. Porém, na maioria das vezes a detecção de *kernel hooking* pode ser feita por programas que executam em nível privilegiado, como por exemplo os *rootkits* [Hoglund and Butler 2005]. No caso geral, a análise de *malware* cuja execução ocorre no nível de usuário é mais confiável caso se aplique a técnica de *kernel hooking*, pois o componente responsável pela captura das ações do *malware* está em um nível de privilégio mais elevado, cujo acesso direto não é permitido. Por outro lado, mesmo com a possibilidade de subversão do *hooking*, os *rootkits* (e programas de nível de *kernel* em geral) precisam ser inicializados por programas de nível de usuário. Portanto, todas as ações executadas durante o processo de inicialização, como por exemplo, o carregamento de um *driver*, são capturadas, podendo ao menos levantar suspeitas sobre um possível comportamento malicioso. Um exemplo de *kernel hooking* comumente utilizado por mecanismos de segurança, como os antivírus, e também por *rootkits* é a técnica de *SSDT hooking*, explicada a seguir. Outro exemplo, explicado adiante, é a técnica de *kernel callbacks*.

SSDT Hooking

O *hooking* da *System Service Dispatch Table*, comumente conhecido como *SSDT hooking*, consiste na modificação de uma estrutura interna presente em sistemas *Windows*, a qual é responsável por armazenar os endereços das *syscalls* do sistema. Esta estrutura é composta basicamente por um vetor de endereços, onde cada índice corresponde a uma das rotinas de chamadas de sistema disponibilizadas pelo sistema operacional, representando uma tabela. Tal tabela reside no *kernel* do sistema operacional e, portanto, programas no nível de usuário não têm acesso a ela. Esta tabela é utilizada pelo sistema operacional quando uma chamada de sistema é requisitada, retornando assim o endereço de memória da função apropriada [Blunden 2009]. Para realizar o *hooking* de SSDT é preciso utilizar um *driver* que opere em nível de *kernel*. Como esse *driver* executa em modo privilegiado, ele pode realizar alterações em outros programas e estruturas internas do sistema presentes no nível de *kernel*. Portanto, o *driver* tem permissão para alterar os endereços contidos na SSDT, trocando-os por valores que indiquem métodos de seu controle. Antes de realizar a troca, os endereços originais precisam ser armazenados para que possam ser utilizados posteriormente, completando assim a requisição feita originalmente. Os endereços alterados irão apontar para funções interceptadas, que serão

executadas ao invés das originais. Como se tem o controle destas funções, é possível monitorar as requisições feitas ao sistema e os valores retornados, ou modificar as respostas retornadas pelo sistema operacional ao programa sob análise. Além disso, fica a cargo das funções controladas realizar a chamada às *syscalls* originais, através dos endereços salvos antes das modificações na SSDT, possibilitando que o fluxo de execução original de um *malware* monitorado seja mantido.

Técnicas de *kernel hooking* são muito mais poderosas do que as de *userland* justamente por atuarem em um nível privilegiado, o que lhes permite maior controle sobre os demais programas que executam no sistema operacional. Outra vantagem é que o monitoramento fica transparente para as aplicações de nível de usuário, pois elas não têm acesso às aplicações que executam no nível do *kernel*. Uma desvantagem deste tipo de técnica diz respeito às informações que são extraídas das funções que se pode interceptar. Na abordagem de IAT *hooking*, é possível capturar funções contidas nas *DLLs* utilizadas por um programa, as quais correspondem muitas vezes ao modo utilizado pelo programador para executar a chamada de sistema. Já no caso das chamadas de sistema capturadas através do SSDT *hooking*, a informação não se apresenta de uma forma tão clara. Por exemplo, caso se queira obter o nome de um arquivo utilizado durante uma *syscall*, pode ser necessário realizar a verificação do conteúdo de outras estruturas utilizadas na invocação da *syscall*, dado que este nome muitas vezes não é passado de forma direta. Neste caso é necessário realizar alguns procedimentos para "traduzir" os dados passados como argumento para a *syscall* de forma a encontrar a informação desejada. Além deste problema, o SSDT *hooking* é dependente da versão do sistema onde ele está sendo aplicado, sendo que para cada versão o SSDT *hooking* deve ser feito de uma maneira diferente. Tal efeito acontece pois podem ocorrer modificações na SSDT entre as versões, fazendo com que endereços antes utilizados para identificar uma *syscall* não sejam os mesmos.

Um sistema de análise de *malware* que faz uso desta técnica é o *JoeBox* [Joebox 2011]. Além de interceptar as chamadas da SSDT ele também realiza um *hooking* de nível de usuário, o que possibilita obter um volume bem maior de informação. Infelizmente só é possível submeter um número limitado de *malware* a este sistema, dado que se trata de um sistema comercial.

3.2.3. *Kernel Callbacks* e *Filter Driver*

Callbacks são funções disponibilizadas pelo sistema operacional que notificam uma aplicação sobre determinadas modificações no sistema, como por exemplo, a criação de uma chave de registro ou de um novo arquivo [Seifert et. al. 2007]. Estas funções são bem documentadas e, portanto, sua implementação não apresenta incompatibilidades entre as diferentes versões do *Windows*, como ocorre no caso do *SSDT Hooking*, que realiza modificações em estruturas do *kernel* específico de cada versão do sistema operacional. Isto torna possível a monitoração de determinadas ações que ocorrem no sistema de uma forma mais simples e genérica, permitindo a identificação de comportamento possivelmente malicioso. Uma limitação presente nesta abordagem é que ela permite somente a captura das ações realizadas por funções disponibilizadas pelo sistema operacional, o que pode levar à obtenção de um comportamento de execução incompleto.

Uma tentativa de melhorar a captura de dados com o uso de *kernl callbacks* é atra-

vés do uso de *filter drivers*. Seu funcionamento é similar ao de um filtro, interceptando todas as requisições feitas a um determinado dispositivo do sistema. Eles se interpõem entre o *driver*, o qual estão sendo interceptadas as requisições, e o nível de usuário, tendo acesso assim a todas as chamadas feitas ao dispositivo interceptado. Seu uso em conjunto com *kernel callbacks* possibilitam que seja capturado um número maior de informações porém, elas ainda são restritas a um limitado conjunto, o qual pode resultar em uma análise incompleta ou inconclusiva. Uma ferramenta que implementa a técnica de *kernel callbacks* e *filter driver* é o CaptureBat [CaptureBat 2011]. Sua utilização requer o carregamento de *drivers* no sistema operacional a ser monitorado, o que possibilita sua aplicação tanto em máquinas virtuais como reais.

3.2.4. *Debugging*

Utilizada inicialmente para fins não maliciosos, a técnica de *debugging* consiste em para execução de um programa em um dado momento, sendo possível verificar o que será executado a seguir. Normalmente ela é utilizada por desenvolvedores de *software*, que desejam encontrar em qual ponto sua aplicação está apresentando problemas. Para isto, basta colocar um marcador em determinados pontos, chamados de *breakpoints*, os quais irão interromper o fluxo de execução do programa no aguardo de um comando sobre o que fazer a seguir. *Debuggers*, programas que executa o processo de *debugging* de uma aplicação, são separados em duas categorias, os que utilizam os recursos providos pelo processador, mais comuns, e os que emulam o processador, sendo estes os mais poderosos pois controlam toda a execução do programa. Se o emulador for bem feito, será muito difícil para um programa descobrir que está sofrendo *debugging*, dado que o *debugger* estará no controle da execução.

Os *breakpoints* podem ser divididos em dois grupos, os *software breakpoints*, que modificam o código da aplicação onde se deseja realizar o *debugging*, e os *hardware breakpoints*, que utilizam recursos do processador, sendo os mais difíceis de serem identificados. Como os *software breakpoints* modificam a aplicação, basta um simples teste de intergridade para identificar a presença de um *breakpoint* no código. Já os *hardware breakpoints* não sofrem deste problema, dado que as modificações feitas para realizar o *debugging* estão em estruturas disponibilizadas pelo processador. Mesmo que o programa tente identificar estas mudanças, o *debugger* pode interceptar estas requisições e retornar uma resposta falsa, não revelando assim que o programa está sofrendo um *debugging*.

Na análise dinâmica de *malware* a técnica de *debugging* pode ser utilizada para identificar características do *malware*. Com isto será possível desenvolver métodos que possam identificar novos *malware* com características semelhantes, facilitando a identificação destes exemplares e evitando que novas contaminações ocorram. Por obter dados de mais baixo nível, composto pelo código *assembly* da aplicação, é preciso realizar algumas inferências para obter informações de mais alto nível, como por exemplo um nome de arquivo. Algumas aplicações possibilitam que o *debugger* tenha acesso direto à essa informação, sendo possível identificar o momento em que a aplicação faz referência a ela.

Duas ferramentas de *debugger* bem conhecidas e utilizadas constantemente, uma paga e outra gratuita, são o *IDA Pro* e o *OllyDBG* respectivamente. A primeira dispõe de mais recursos que podem auxiliar durante o processo de *debugging*, como por exemplo

um gráfico que mostra todas as chamadas de funções identificadas no programa. Com ele é possível ver a relação existente entre as funções presentes no programa. Já a segunda, *OllyDBG*, mesmo sendo uma ferramenta gratuita dispõe de recursos indispensáveis durante o processo de *debugging*, como por exemplo uso de *plugins* que podem ajudar no processo de análise.

3.2.5. Engenharia Reversa

O processo de engenharia reversa consiste em extrair informações sobre um *software*, de forma que seja possível compreender seu funcionamento. Não existe uma ferramenta que realize de forma automática este processo, sendo somente possível realizá-lo de forma manual. Normalmente é aplicado em *software* onde não é possível se ter acesso ao código fonte, possibilitando assim um maior entendimento sobre as ações executadas pelo *software* durante sua execução, auxiliando inclusive na remontagem do código do programa.

Na análise de *malware*, esta técnica é utilizada quando se deseja descobrir, de forma detalhada, quais os passos desenvolvidos pelo *malware* no sistema. Isso possibilita, por exemplo, identificar quais técnicas ele utilizou para comprometer o sistema, como ele esconder suas atividades para ocultar sua execução, quais dados do sistema comprometido ele captura, dentre outras atividades normalmente executadas por *malware*. A partir dessas informações, é possível inclusive criar procedimentos que retirem da máquina comprometida todas as modificações feitas pelo *malware*, levando-a a um estado integro, anterior ao comprometimento. Outra aplicação para a engenharia reversa de *malware* é o reconhecimento de rotinas de cifração presentes nele. Elas normalmente são aplicadas ao *malware* para evitar que certos dados fiquem em claro no código binário, como por exemplo um endereço utilizado pelo *malware* onde está outro componente utilizado por ele, o qual será obtido durante sua execução, ou um endereço de *email* para onde serão enviados os dados capturados na máquina comprometida.

Para que uma engenharia reversa seja realizada, é preciso um bom nível de conhecimento sobre o ambiente onde a aplicação irá executar, para compreender as iterações realizadas entre o programa e o sistema operacional bem como as respostas retornadas em cada uma delas. Com a mescla de técnicas que combinam dados de baixo e alto nível, é possível obter informações necessárias para realizar uma engenharia reversa de forma fácil e rápida. Portanto, todas as técnicas de análise dinâmica apresentadas até aqui podem ser aplicadas durante o processo de engenharia reversa, evitando que seja desperdiçado tempo na busca de informações que são providas naturalmente por elas.

3.3. Sistemas de Análise Dinâmica

As técnicas apresentadas no capitulo anterior são utilizadas em um grande número de sistemas de análise de *malware* e *Web malware*. Normalmente, os sistemas apresentam os dados coletados de modo que seja fácil compreender as ações executadas pela amostra durante a análise. Alguns sistemas aceitam somente submissão através de seus *Web sites* enquanto outros só podem ser executados se instalados em uma máquina local. Neste capitulo serão discutidos em cada subseção sistemas de análise relevantes, frequentemente utilizados na análise de *malware* e *Web malware*.

3.3.1. Sistemas de análise de *malware*

3.3.1.1. *Anubis*

Iniciou suas atividades em 2007, baseado em um trabalho de mestrado, *TTAnalyze* [Bayer et. al. 2006]. Utiliza a técnica de *Virtual Machine Introspection* (VMI) aplicada ao *Qemu*, explicada detalhadamente no capitulo anterior, para capturar as ações executadas pelo *malware* no sistema de análise. O sistema operacional utilizado no ambiente de análise é um *Windows XP* SP3, com uma instalação básica. Ao final da análise, que pode demorar 8 minutos, é produzido um relatório em vários formatos, *html*, *xml* e txt, que sumariza as ações do *malware*. Além disto, caso seja gerado algum tráfego de rede, um arquivo no formato pcap também é fornecido. A submissão pode ser feita em lote, através de um *script python*, ou de forma singular, onde somente um *malware* é enviado. No caso desta submissão, é possível torna-la mais rápida, passando-a à frente das submissões em lote. Para isto, basta que um *captcha* seja fornecido no momento da submissão.

Apesar do uso de VMI para capturar as ações desenvolvidas pelo *malware*, o *Anubis* têm dois componentes no sistema de análise, um *driver* e um programa de nível de usuário, os quais irão capturar os valores contidos no registrador *CR3*, usados pelos processos do *malware*, e realizar a comunicação entre o ambiente de análise e a máquina *host* respectivamente. A tabela 3.2 mostra as vantagens e desvantagens presentes no sistema *Anubis*, levado em consideração todos os pontos levantados até aqui.

Tabela 3.2. Vantagens e desvantagens do sistema de análise *Anubis*.

Vantagens	Desvantagens
A captura é feita por um componente fora do ambiente de análise.	Existem componentes do sistema de análise dentro do ambiente de análise.
Não é necessário nenhum preparo local para executar a análise.	Caso o sistema esteja fora do ar, a análise não pode ser realizada.

3.3.1.2. *CWSandbox*

Teve início em 2007, utilizando a técnica de *userland hooking* para obter as informações geradas pelo *malware*, em máquinas virtuais. Os ambientes de análise têm como sistema operacional um *Windows XP* [Willems et. al. 2007]. A captura das informações é realizada por uma *DLL (Dynamic Link Library)*, que precisa ser injetada no processo do *malware*. Quando ela é carregada, as principais funções utilizadas para fazer a interface entre o programa e o sistema de análise, como por exemplo modificações em arquivos, têm seu início modificado, de forma que um desvio incondicional seja executado assim que a função é chamada. Finalizada a análise, é gerado um relatório nos formatos *html*, *xml* e txt, contendo as ações realizadas no ambiente pelo *malware*. O processo de submissão pode ser feito de forma singular, para um único arquivo, ou na forma de um arquivo comprimido, contendo múltiplos arquivos para análise. É necessário fornecer um endereço de *email* para onde será enviado o *link* para o resultado da análise, quando este estiver pronto.

Para iniciar o processo de análise, existe um componente dentro do ambiente de análise, o cwsandbox.exe, que irá começar o processo do *malware* em estado suspenso, injetar a *DLL* e retomar a execução do processo. Além disto, este componente será informado caso o *malware* inicialize ou modifique algum processo, para que a *DLL* seja injetada neles também. Existe um outro componente dentro do ambiente de anpalise, que é responsável pela proteção dos componentes de captura, escondendo evidências de sua existência. Na tabela 3.3 é possível ver as vantagens e desvantagens presentes no sistema *CWSandbox*.

Tabela 3.3. Vantagens e desvantagens do sistema de análise *CWSandbox*.

Vantagens	Desvantagens
Os dados capturados são de funções de mais alto nível.	Caso o sistema esteja fora do ar, a análise não pode ser realizada.
Não é necessário nenhum preparo local para executar a análise.	Existem componentes do sistema de análise dentro do ambiente de análise.
Existem componentes que dificultam a detecção do sistema	

3.3.1.3. *Cuckoobox*

Começou como um projeto do *Google Summer of Code* de 2010 ligado à *The Honeynet Project* sendo disponibilizado somente em 2011, tendo seu código totalmente disponível para *download*. Em 2011, foi novamente selecionado para o *Google Summer of Code*, onde foram feitas modificações na técnica de captura de informações e criado um novo componente que esconde os elementos do *Cuckoobox* que estão no ambiente de análise. Utiliza a técnica de *inline hooking* para interceptar as chamadas de sistema executadas pelo *malware*. Não dispõe de uma interface de submissão, sendo que para realizar uma análise é preciso preparar o ambiente antes, instalando localmente todos os requisitos necessários. Não apresenta restrições quanto a versão do sistema que pode ser utilizado no ambiente de análise, podendo ser qualquer versão do *Windows* a partir do XP. Depois de instalado o ambiente e todas as dependências do *Cuckoobox* é preciso inicializar um *script* em *python*, o qual irá carregar todas as configurações necessárias para realizar uma nova análise. Após isto é possível então invocar um outro *script* em *python* que irá enviar o *malware* para análise. Ao final dela, estarão disponíveis em uma pasta os traços de execução criados pelos processos gerados durante a análise, os *snapshots* de telas geradas, arquivos modificados/criados/deletados e o tráfego de rede.

Para implementar o *inline hooking* o *Cuckoobox* precisa carregar uma DLL no processo que deseja monitorar. O *hooking* é implementado de forma diferente em cada função interceptada diferentemente de um simples salto incondicional no início da função. Isto dificulta métodos triviais de detecção, evitando assim que a análise não seja bem sucedida. Atualmente, existem dois métodos implementados, os quais são escolhidos de forma aleatória no momento que o *inline hooking* é instalado. Esta *DLL* é carregada por um *script* em *python* que fica em execução no ambiente de análise durante todo o processo. Além desta tarefa, ele também é notificado caso o *malware* modifique ou crie

um novo processo durante a análise, indicando ao *script* que a DLL de monitoração deve ser carregada nestes outros processos. A tabela 3.4 explicita as vantagens e desvantagens presentes no sistema *Cuckoobox*.

Tabela 3.4. Vantagens e desvantagens do sistema de análise *Cuckoobox*.

Vantagens	Desvantagens
Os dados capturados são de funções de mais alto nível.	Existem componentes do sistema de análise dentro do ambiente de análise.
O código está disponível e é possível customizar o sistema de acordo com suas necessidades.	É necessário preparar o ambiente para executar a análise.
Existem componentes que dificultam a detecção do sistema	

3.3.1.4. *Ether*

Foi apresentado em 2008 e teve seu código disponibilizado em 2009. Utiliza *Virtual Machine Introspection* (VMI), aplicada ao *Xen*, um *hypervisor open source* bem conhecido, para capturar as informações geradas. Para realizar uma análise é preciso instalá-lo localmente, em uma máquina com processador com suporte *Intel VT*, um conjunto de instruções que facilita a virtualização de instruções x86. Terminada a instalação, é necessário preparar o ambiente de análise, instalando um *Windows XP* com SP2 sem nenhum programa adicional. O controle da análise é feito inteiramente por um componente fora do ambiente de análise, o que evita possíveis detecções do sistema. Sempre que for realizar uma nova análise é preciso utilizar este componente, o qual irá enviar o *malware* para o ambiente de análise e executá-lo. Infelizmente, a captura das ações realizadas se restringe a todos os processos do sistema ou somente ao processo do *malware*. Portanto, caso o *malware* crie ou modifique outros processos, o *Ether* não será capaz de restringir a captura somente à estas ações. A tabela 3.5 mostra as vantagens e desvantagens presentes no sistema *Ether*, baseados nos pontos levantados até aqui.

Tabela 3.5. Vantagens e desvantagens do sistema de análise *Ether*.

Vantagens	Desvantagens
A captura é feita por um componente fora do ambiente de análise.	Só faz a monitoração de um processo por vez.
O código está disponível e é possível customizar o sistema de acordo com suas necessidades.	É necessário preparar o ambiente para executar a análise.
Não têm componentes dentro do sistema de análise	É necessário indicar o nome/endereço do processo que se deseja monitorar.

3.3.1.5. *Joebox*

Iniciou suas atividades em 2007, disponibilizando de forma gratuita análises de *malware*. No ano de 2011, tornou-se um serviço pago, aceitando submissões somente mediante pagamento de uma taxa mensal. Utiliza técnicas de *hooking* para obter as ações executadas no ambiente de análise. Fornece relatórios em formato *html* e *xml* junto com os arquivos criados, capturas de tela, tráfego de rede e *dumps* de memória gerados durante a análise. Não há restrições quanto ao sistema de análise que deve ser utilizado, podendo ser qualquer versão *Windows*, acima do XP. Além disto, ele pode ser executado em diversas plataformas como máquinas virtuais, emuladas e reais.

Combinando várias técnicas de *hooking*, tanto de *userland* quanto de *kernel*, o *Joebox* é capaz de obter informações mais detalhadas, capturando dados tanto de baixo quanto de alto nível. Existe um componente interno no sistema de análise, joesandbox-control.exe, que contém várias funcionalidades, dentre elas a inicialização do *malware*. Na tabela 3.6 é possível verificar as vantagens e desvantagens presentes no sistema.

Tabela 3.6. Vantagens e desvantagens do sistema de análise *Joebox*.

Vantagens	Desvantagens
Pode ser executado em diversas arquiteturas.	Não está disponível gratuitamente.
Implementa várias técnicas de captura diferentes.	Existem componentes do sistema de análise dentro do ambiente de análise.

3.3.2. Sistemas de análise de *Web malware*

3.3.2.1. JSand

JSand é um sistema de análise de *Web malware*, de baixa interatividade, que foi apresentado em [Cova et. al. 2010] e pode ser usado através da interface pública de submissão *online*[10]. Sua principal função é analisar o código JavaScript presente na página, provendo informações a respeito de sua execução e informando se a página analisada é benigna, suspeita ou maliciosa. Essa identificação é feita por meio da detecção de anomalia no comportamento do código JavaScript. Para emular a página a ser analisada, o JSand utiliza uma versão modificada do HtmlUnit, uma plataforma em Java para testes de aplicações *Web*. Como os *Web malware* atualmente atacam vulnerabilidades de diversas aplicações, o sistema emula todo objeto ActiveX requisitado pelo código, de forma que uma verificação pela presença dele retornará positiva e o código continuará sua execução.

Para realizar a detecção por anomalia, o sistema JSand extrai dez atributos a partir da análise realizada. Esses atributos representam características de redirecionamento, ofuscação, preparação do ambiente para o abuso de vulnerabilidades e o processo de abuso. O treinamento e detecção são realizados com o uso da ferramenta libAnomaly[11], desenvolvida pelo mesmo grupo que desenvolveu JSand.

[10] http://wepawet.cs.ucsb.edu
[11] http://www.cs.ucsb.edu/~seclab/projects/libanomaly/

As informações disponibilizadas após a análise dizem respeito ao comportamento do código JavaScript presente na página e incluem trechos de código desofuscados, vulnerabilidades das quais o código tenta abusar, *shellcodes* utilizados, objetos ActiveX utilizados, *links* para o sistema Anubis com a análise de arquivos executáveis que o código tenha tentado obter e requisições HTTP. A tabela 3.7 apresenta as vantagens e desvantagens da abordagem usada por esse sistema.

Tabela 3.7. Vantagens e desvantagens do sistema de análise *JSand*.

Vantagens	Desvantagens
Por ser emulada, a análise é executada rapidamente.	O ambiente emulado falha ao analisar certos códigos.
Pode detectar ataques a vulnerabilidades desconhecidas.	Pode detectar apenas ataques que utilizam JavaScript.

3.3.2.2. PhoneyC

O sistema PhoneyC, apresentado em [Nazario 2009], é um *honeyclient* de baixa interatividade que também utiliza um emulador para processar as páginas analisadas e possui seu código disponível para *download*. Ele é capaz de analisar códigos JavaScript e Visual Basic Script. A emulação do ambiente de JavaScript é feita com o uso do interpretador da Mozilla Foundation, SpiderMonkey, que faz parte do navegador *Web* Firefox. Já o código Visual Basic Script é primeiramente transformado em um código equivalente em Python, pela ferramenta vb2py[12], e então processado pelo próprio interpretador da linguagem.

Para detectar tentativas de abuso de vulnerabilidades o sistema emula certos componentes vulneráveis. PhoneyC prove informações sobre objetos ActiveX utilizados, vulnerabilidades que o código tenta atacar e *shellcodes* utilizados. A tabela 3.8 apresenta as vantagens e desvantagens desse sistema.

Tabela 3.8. Vantagens e desvantagens do sistema de análise *PhoneyC*.

Vantagens	Desvantagens
Por ser emulada, a análise é executada rapidamente.	O ambiente emulado falha ao analisar certos códigos.
Além de JavaScript, detecta ataques que utilizam Visual Basic Script	Não pode detectar ataques a vulnerabilidades desconhecidas.

3.3.2.3. Capture-HPC

O sistema Capture-HPC, descrito com mais detalhes em [Seifert and Steenson 2006], é um *honeyclient* de alta interatividade que possui seu código disponível. As páginas a serem analisadas são processadas dentro de um ambiente virtualizado, utilizando um navegador *Web* completo e um *driver* de *kernel* que captura as chamadas de sistemas realizadas

[12] http://vb2py.sourceforge.net

pelo navegador. Caso essas chamadas de sistema sejam consideradas anômalas, a página analisada é classificada como maliciosa.

Após a análise o sistema informa as chamadas de sistema que foram identificadas como anômalas. As vantagens e desvantagens desse sistema podem ser vistas na tabela 3.9.

Tabela 3.9. Vantagens e desvantagens do sistema de análise *Capture-HPC*.

Vantagens	Desvantagens
Utiliza um navegador *Web* completo para processar as páginas	Devido ao uso de ambiente virtualizado, a análise é mais demorada.
Pode detectar ataques a vulnerabilidades desconhecidas.	Pode detectar apenas ataques bem sucedidos, que resultem em chamadas de sistema anômalas.
Pode detectar ataques independentemente da linguagem utilizada.	

3.4. Caixa de Ferramentas

3.4.1. Informações Gerais

Neste capítulo será apresentado um conjunto de ferramentas gratuitas e disponíveis na Internet que são capazes de criar um ambiente de análise dinâmica de *malware*. Além disso, serão apresentadas ferramentas para análises mais aprofundadas, *debugging* e forense computacional. O ambiente de análise a ser implementado neste capítulo será denominado de Protótipo Sandbox. Como base para sua construção, será usada uma máquina virtual para agilizar a restauração do ambiente. É importante deixar claro algumas limitações do sistema, visto que em sua concepção foi dado enfoque à praticidade na montagem e no uso. Dessa forma o sistema possuirá pontos vulneráveis, que serão destacados no decorrer do texto. As ressalvas são a necessidade de um analista para dar confiabilidade aos relatórios providos e a falta de proteção contra técnicas de detecção de ambientes virtuais.

3.4.2. Modelo proposto para o sistema de análise

É importante identificar e organizar a ordem com que as ferramentas deverão ser executadas. Dessa forma, pode-se tornar a análise mais eficaz. Na Figura 3.5 pode-se observar que o conhecimento sobre o funcionamento das ferramentas é fundamental para uma boa análise, pois a ordem em que os comandos foram executados altera diretamente a qualidade das informações obtidas. No trecho 1, caso o *malware* contenha algum *packer* que ofusque o código, a extração das *strings* dificilmente trará informações úteis, enquanto no trecho 2, verifica-se se foi utilizado o *packer* UPX. Caso este tenha sido utilizado, é removido do *malware*[13], assim a ferramenta *strings* conseguirá capturar dados relevantes que estiverem no código.

O Protótipo Sandbox possui dois ambientes,o primeiro é o hospedeiro, no qual haverá um sistema operacional base, para processar as informações obtidas da análise dinâmica. Este ambiente é a área de trabalho principal do analista e não deve ser conta-

[13] A mesma ferramenta que empacota um binário com o *packer* UPX pode ser usada para removê-lo.

```
/* Trecho 1: ineficiente para extração de dados */
relatorio.escrever(strings(malware));
relatorio.escrever(packer(malware));

/* Trecho 2: eficiente para extração de dados */
nomePacker = packer(malware);
if(nomePacker == "upx")
    malware = unpacking(malware, nomePacker);
relatorio.escrever(strings(malware));
relatorio.escrever(nomePacker);
```

Figura 3.5. Pseudo-código que demonstra a importância do uso das ferramentas na ordem correta

minada pelo *malware*, ficando responsável apenas pela análise estática e por hospedar as máquinas virtuais.

O segundo ambiente é o virtualizado. Nele os exemplares de *malware* serão executados para que seu comportamento seja capturado. Portanto, esses ambientes deverão ser preparados para estimular e monitorar as atividades maliciosas e, posteriormente, enviar as informações coletadas para a máquina hospedeira. É importante que a máquina virtual não esteja na mesma rede que outros sistemas, pois pode haver contaminação destes. A arquitetura projetada para o Protótipo Sandbox é ilustrada na Figura 3.6.

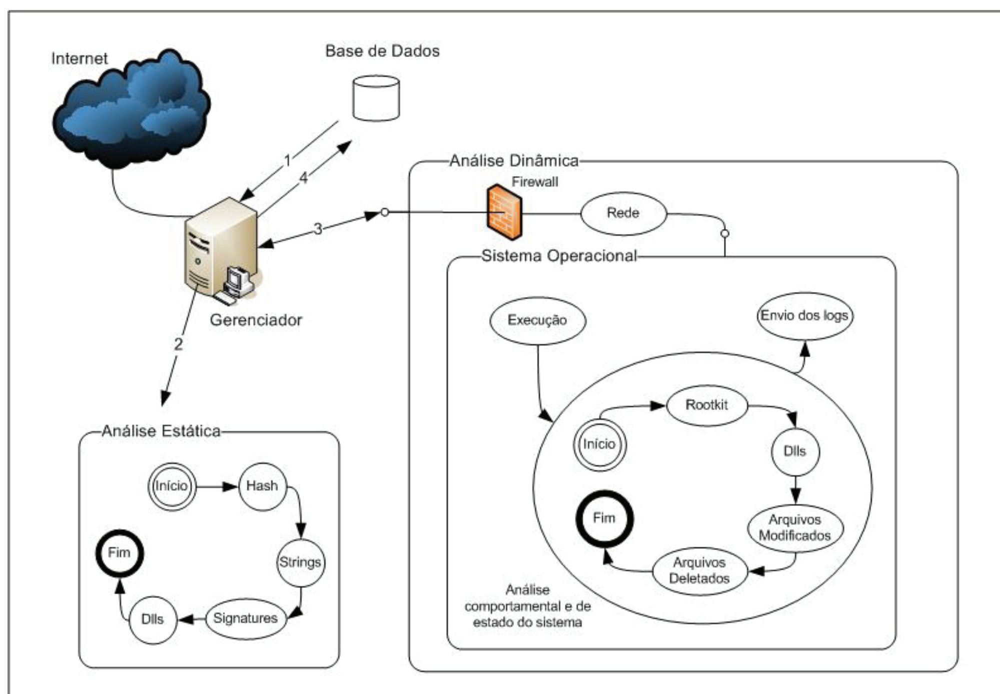

Figura 3.6. Arquitetura do Protótipo Sandbox

Na Figura 3.6, observa-se que o ambiente base (Gerenciador) opera como uma camada intermediária entre a Internet e o ambiente de análise (Análise Dinâmica) virtu-

alizado. Isto torna possível restringir o acesso da máquina virtual através de um *firewall* entre os ambientes, além de bloquear tentativas de interação com a máquina real. Outra responsabilidade importante do Gerenciador é o armazenamento dos relatórios em uma base de dados, podendo ser feito de forma simples, como um diretório, ou algo mais complexo como um banco de dados.

3.5. Máquina base

Para os testes com o Protótipo Sandbox foi utilizado um computador com processador Intel Core 2 Duo de 1.6 GHz, 2GB de memória RAM e disco rígido de 160 GB. Outras configurações que sejam equivalentes ou superiores também são válidas. A sugestão é apenas uma estimativa do mínimo que o computador precisa ter para realizar análises sequenciais sem *overheads* prejudiciais. Esta máquina é a que será usada para armazenar os resultados dos relatórios, efetuar a análise estática, hospedar a máquina virtual e gerenciar a análise dinâmica. O sistema operacional escolhido para a máquina base foi um Ubuntu linux com sua instalação padrão.

3.5.1. Base de resultados

A base de resultados do Protótipo Sandbox terá como finalidade armazenar *malware* e todos os arquivos produzidos durante as análises, inclusive o relatório final. Para isto, será construída em um diretório normal do sistema, terá como chaves primárias o hash sha256 do *malware* e será gerenciada através de shell script.

A base de resultados contém diversos diretórios, cada um com a coletânea de arquivos relacionados ao *malware*, tais como o arquivo executável, o relatório e os arquivos auxiliares produzidos nas análises. A organização da base de resultados pode ser melhor visualizada na Figura 3.7.

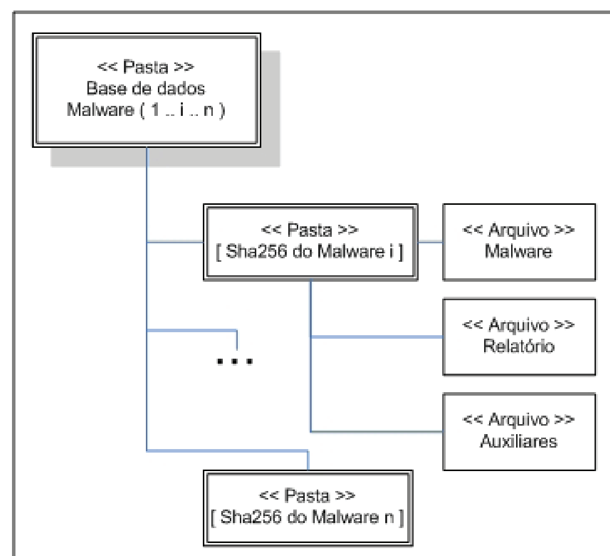

Figura 3.7. Estrutura de pastas para armazenamento dos arquivos produzidos pelas análises

Existem algumas operações que são fundamentais para gerenciar a base de dados a

ser construída, como a busca, a exclusão e a verificação se existe uma análise. Estas ações podem ser implementadas com comandos de *shell script*, como o exemplo mostrado na Figura 3.8.

```
# Configuração dos parâmetros
sha256=`sha256sum ${malware} | cut -d “ ”-f 1`

# Verificando se análise existe
if [ -d ${baseDados}/${sha256} ]
then
     echo "Malware já foi analisado!"
     nautilus ${baseDados}/${sha256} &
else
     echo "Malware ainda não analisado!";
     rodarAnalise
fi
```

Figura 3.8. Comandos de shell script usados para verificar se um *malware* já foi analisado

3.5.2. Análise estática simplificada

A análise estática tem como objetivo coletar informações presentes no *malware* sem ter que executá-lo. No Protótipo Sandbox as informações foram divididas em quatro grupos que podem ser visualizados na Figura 3.9.

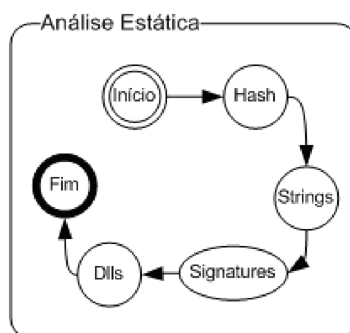

Figura 3.9. Informações obtidas na análise estática

Após o início da análise estática, a primeira informação buscada é uma chave que identifique univocamente o *malware*, a opção escolhida foi a função de *hash* criptográfico `sha256`, todavia poderia ser qualquer outra, como `md5` (mais comum) ou *sha512*.

Posteriormente, é utilizado o programa `strings` para buscar cadeias de caracteres relevantes, com pelo menos 3 *bytes*, no código. Assim, recomenda-se o uso de filtros com expressões regulares e a remoção prévia do *packer*, como mostrado na Seção 3.4.2. Mesmo que o *packer* não seja removido, informações sobre as bibliotecas utilizadas estarão presentes e serão úteis para indicar as ações do *malware* e caminhos para um processo de engenharia reversa, como pode ser visto na Figura 3.10.

Algumas identificações do *malware* podem ser coletadas com o uso de programas antivírus, identificadores de *packer* e identificadores de bibliotecas. No Protótipo

Exemplo de strings de malware com Armadillo v1.71
RegSetValueExA, RegCloseKey, RegOpenKeyExA, CryptGetHashParam,
CryptDestroyHash, CryptReleaseContext, CryptHashData,
CryptCreateHash, CryptAcquireContextA, AdjustTokenPrivileges,
LookupPrivilegeValueA, OpenProcessToken, RegCreateKeyA,
RegQueryValueExA, GetStartupInfoA, system32, SeDebugPrivilege, ...

Figura 3.10. Exemplo de *strings* encontradas em um *malware* com o *packer Armadillo*

Sandbox foram utilizados apenas identificadores de *packer* e de bibliotecas, através dos programas `pefile` e `sigcheck`. É interessante saber o *packer* utilizado para tentar removê-lo, e também saber se a análise dinâmica poderá ser feita, visto que certos *packers*, como o tElock e o Themida, são capazes de identificar ambientes emulados. O programa *sigcheck* é importante para verificar se as bibliotecas nativas utilizadas são legítimas.

Caso alguma biblioteca seja falsa, pode-se extrair informações usando os programas *objdump* e *Dependency Walker*, que revelaram dados do cabeçalho, seções e funções da biblioteca. Para a análise estática automatizada é preferível o programa `objdump`, por permitir seu uso através de linha de comando.

3.5.3. Máquina virtual

A máquina virtual tem como finalidade prover um ambiente de análise controlado para o *malware* ser executado, isolando a máquina base, além de possibilitar a rápida restauração do ambiente após o comprometimento. Para o Protótipo Sandbox foi escolhido o programa Virtualbox como tecnologia de virtualização.

3.5.4. Análise dinâmica

Na análise dinâmica é preparado um ambiente para monitorar todas as atividades do *malware* que será executado, obtendo assim seu comportamento e a interação dele com o sistema. No caso do Protótipo Sandbox as ações do *malware* foram divididas em rede, arquivos modificados e excluídos, registros modificados e excluídos, *rootkits*, processos criados e finalizados e DLLs. O processo de análise dinâmica pode ser visto na Figura 3.11.

3.5.4.1. Preparação do ambiente virtual

O ambiente virtualizado foi configurado com um disco de expansão dinâmica e 256 MB de memória RAM, e foi instalado com a configuração padrão do Windows XP Service Pack 3.

Para iniciar a máquina virtual é utilizado um conjunto de *scripts*, tanto na máquina virtual como na máquina base. A máquina base fica encarregada por iniciar a máquina virtual e depois desligar forçadamente a mesma, após o tempo limite de análise. A Figura 3.12 mostra os comandos utilizados para iniciar e finalizar a máquina virtual.

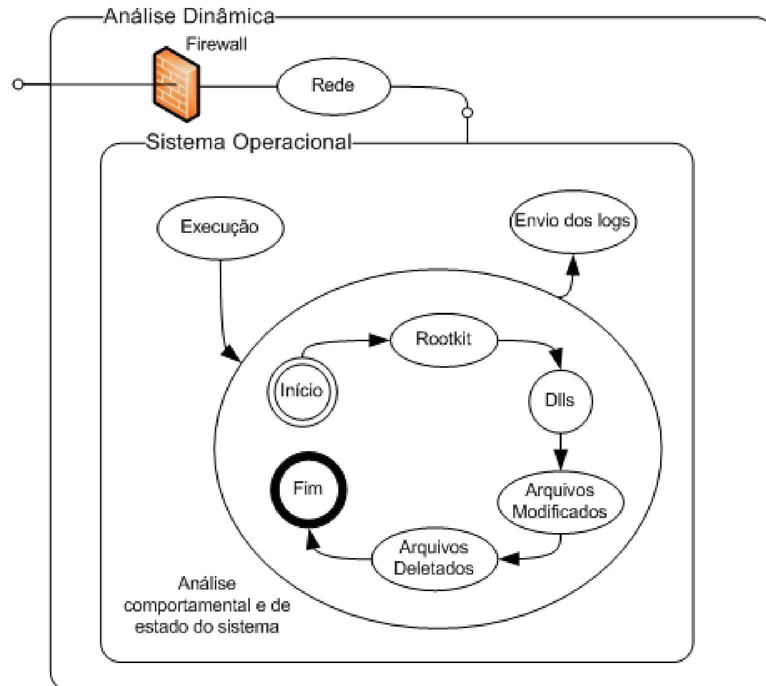

Figura 3.11. Processo de análise dinâmica

```
# Iniciar a máquina virtual
VBoxManage -q startvm ${MACHINE}

# Desligar a máquina virtual
VBoxManage -q controlvm ${MACHINE} poweroff

# Desligar forçadamente a máquina virtual
killall VirtualBox
killall VBoxXPCOMIPCD
```

Figura 3.12. Comandos para inicialização e término de máquinas virtuais

A máquina virtual é iniciada a partir de um *snapshot*[14], no qual o sistema está completamente inicializado e com um *script* do tipo WSH (Windows Script Host) sendo executado. Esse *script* tem a função de obter o *malware* através de FTP, iniciar/executar as ferramentas de análise e executar o *malware*, como observado na figura 3.13.

3.5.4.2. Monitoração do sistema

Para a monitoração do sistema é utilizada a ferramenta `Capture-BAT` [CaptureBat 2011], que é capaz de monitorar certas operações feitas pelo *malware*. Essas operações

[14] *Snapshots* são arquivos que guardam o estado do sistema. Com ele é possível iniciar o sistema virtualizado a partir de um estado previamente salvo

```
' Jogando os arquivos da Real para a Virtual
WshShell.Run "ftp -s:C:\CaixaCBT\ftp_m.txt <IP>",1,true
WScript.Sleep 2000

' Ligar CaptureBAT
WshShell.Run "C:\CaixaCBT\Capture\CaptureBAT.exe -cn -l
log_CBT.txt",2,false
WScript.Sleep 2000

' Ligar Malware
WshShell.Run "C:\CaixaCBT\Malware\malware.exe",2,false
WScript.Sleep 2000

' Deixar o Malware Rodando
For IntLoop = 1 To 1
    WScript.Sleep 60000
Next
```

Figura 3.13. *Script WSH* para inicialização da ferramenta de análise e execução do *malware* no ambiente virtualizado

incluem criação, modificação e remoção de arquivos e registros, além da inicialização e término de processos. São usados filtros para monitorar apenas as ações do *malware* e de processos iniciados por ele.

3.5.4.3. Tráfego de rede

O tráfego de rede pode ser monitorado de duas formas, internamente à máquina virtual, através do programa `Capture-BAT`, ou na máquina base, com o programa `tcpdump`, utilizando filtros. Por meio do tráfego de rede é possível descobrir servidores de distribuição de *malware*, contas de email utilizadas pelo criador do artefato, tentativas de ataque a outros sistemas e sistemas de comando e controle utilizados para passar comandos ao *malware* caso seja do tipo *bot*.

3.5.4.4. Rootkit

Alguns exemplares de *malware* mais sofisticados utilizam *rootkits* para que as ações maliciosas sejam executadas em nível de *kernel* e, portanto, não possa ser monitorada por ferramentas que capturam as ações dentro do ambiente virtualizado, como o `Capture-BAT`. Dessa forma é preciso, pelo menos, identificar quando os *rootkits* estão sendo usados. A ferramenta `Gmer` é utilizada para essa tarefa

3.5.4.5. DLLs

As bibliotecas encontradas na análise estática não são necessariamente as mesmas utilizadas durante a execução do *malware*, assim é preciso checar quais foram carregadas em tempo de execução. Para essa tarefa utiliza-se a ferramenta `ListDlls`.

3.5.4.6. Término da análise dinâmica

Ao fim da análise, o *script WSH* de controle deverá terminar os processos do *malware* e das ferramentas utilizadas, reunir todos os arquivos de registro e enviá-los por FTP para a máquina base. Esta, por sua vez, deverá conferir se os arquivos de registro foram enviados corretamente e deve desligar a máquina virtual forçadamente, caso precise, como mostrado na Seção 3.5.4.1. Por fim, a máquina utilizada será restaurada para o último *snapshot*. Os diversos arquivos de resultados serão processados pela máquina base e reunidos em um único relatório, para então ser armazenado na base de dados.

3.5.5. Vulnerabilidades do sistema

Como enunciado na Seção 3.4.1, o Protótipo Sandbox possui diversas vulnerabilidades, como por exemplo, não ter proteção contra rotinas que identifiquem ambientes virtualizados. Outra fragilidade é a identificação de uma das ferramentas utilizadas na análise dinâmica. Assim, caso o relatório seja suspeito é preciso realizar uma análise manual e aprofundada no *malware*.

3.5.6. Análise aprofundada

Para análises mais aprofundadas é preciso compreender em baixo nível o que o *malware* está fazendo no sistema, para isto, é possível fazer um *dump* da memória RAM em um momento crítico para analisá-lo ou fazer *debugging* do código binário. Através de um *dump* é possível identificar interceptações, *rootkits* e mutexes (objetos de sincronização) presentes no sistema e que podem estar bloqueando a monitoração de alguma atividade maliciosa, além de ser possível ver *strings*, bibliotecas, conteúdo da pilha (*stack*) e conteúdo da *heap* dos processos que estavam em execução. Uma opção de programa para essa finalidade é o `Memoryze`. Para realizar a engenharia reversa e o compreendimento profundo do funcionamento do *malware*, é preciso utilizar um *debugger*, como o *OllyDbg*. Como a análise manual do código é uma tarefa exaustiva, recomenda-se utilizá-la apenas para conferir os trechos mais suspeitos.

3.6. Estudo de Caso

Nesta parte será apresentado um estudo de caso do processo completo de comprometimento de uma máquina, que ocorreu a partir do acesso a uma *URL* maliciosa. Este endereço foi retirado do site *www .malwaredomainlist.com* que disponibiliza *URL*s maliciosas diariamente. A análise foi feita utilizando um sistema de análise de *URL*s maliciosas, desenvolvido por um dos autores, onde todas as ações desenvolvidas pela *URL* são inspecionadas, juntamente com as *system calls* realizadas pelo processo do *browser* e o tráfego de rede gerado. Serão apresentados alguns trechos dos relatórios produzidos pelo sistema, bem como uma explicação sobre as ações que estão sendo observadas. O exemplar analisado utiliza um *applet java* malicioso, o qual explora uma vulnerabilidade e possibilita o *download* de um *malware*, que é executado no sistema logo após a exploração.

A arquitetura do sistema onde foi realizado o estudo é composto de um *Desktop* com uma máquina virtual do sistema *VirtualBox*, dispondo de acesso controlado à internet. O sistema utilizado no ambiente *guest*, ou seja o ambiente virtualizado, é um

Windows XP SP2 com instalação básica. Para permitir a execução de certos tipos de código que não dispõem de suporte nativo em sistemas *Windows* foi necessário adicionar alguns aplicativos extras, como por exemplo *Java* e *Flash*. A *URL* maliciosa foi acessada em um *browser Internet Explorer 8*, normalmente utilizado pela grande maioria de usuários de sistema *Windows*.

Primeiramente, o *Internet Explorer* é iniciado juntamente com o componente responsável pela captura das ações desenvolvidas pela *URL* e pelo que faz a captura das chamadas de sistema executadas pelo processo do*browser* e seus filhos. Após esta etapa, o *browser* fica a espera de uma *URL*, para que então possa ser realizada uma análise. Diferentemente do componente que obtém as informações de execução da *URL*, o componente de captura de chamadas de sistema fica capturando todas as ações do *Internet Explorer*, mesmo que a *URL* não tenha sido carregada ainda. As ações desnecessárias serão filtradas posteriormente, deixando somente as ações relevantes. Terminada a etapa de instanciação do sistema, a *URL* maliciosa é passada para o *browser*, o qual irá acessá-la, tendo assim todas as atividades geradas capturadas.

Quando a página utilizada no estudo de caso é carregada pelo *browser*, o componente de captura obtém várias ações inofensivas, realizadas normalmente por qualquer página *web*. No meio destas ações pode haver código de exploração, caso algum *script* malicioso em JavaScript seja executado. No trecho de código apresentado a seguir 3.1, uma página HTML é utilizada para chamar um objeto Java que recebe um parâmetro (linha 4) cujo valor, representado por uma sequência de caracteres "A", é usado para explorar uma vulnerabilidade na máquina virtual Java que executa no *browser* do lado do cliente.

Trecho de Código 3.1. Código HTML utilizado na exploração do *browser*

```
1 <html>
2     <object id="java_obj" classid="clsid:CAFEEFAC-DEC7-0000-0000-
        ABCDEFFEDCBA" width="0" height="0">
3             <PARAM name="launchjnlp" value="1"/>
4             <PARAM name="docbase" value="AAAAAAAAAAAAAAAAAAAA..."/
                >
5     </object>
6 </html>
```

Observando a saída gerada pela ferramenta que captura as ações executadas durante o carregamento da página, mostrada no trecho de código 3.2, podemos observar que um dos atributos presentes na linha 2, e os das linhas 3 e 4 do trecho de código 3.1 estão presentes nas ações capturadas pela ferramenta de análise. É possível identificar isto nas linhas 1, 2 e 3 do trecho de código 3.2, que correspondem a uma parte do código obtido durante a execução da pagina maliciosa no *browser*.

Trecho de Código 3.2. Ações capturadas durante o carregamento da *URL* maliciosa.

```
1 10:27:45.010 <URL MALICOSA> SET PROPERTY clsid=CAFEEFAC-DEC7-0000-0000-
    ABCDEFFEDCBA
2 10:27:45.070 <URL MALICOSA> SET PROPERTY launchjnlp=1
3 10:27:45.230 <URL MALICOSA> SET PROPERTY docbase=<EXPLOIT>
```

Já nas ações desenvolvidas pelo processo do *browser* e seus filhos, foi possível notar a ação de um *applet java* e de vários outros processos. Cada ação executada é

colocada em uma linha dividida em três campos, sendo o primeiro o nome do processo executor da ação, o segundo o tipo de ação realizada e o terceiro o alvo da ação. Dentre as ações executadas pelo processo do *browser* foi possível identificar um conjunto de ações relevantes, que correspondem à criação de arquivos, de um *applet java* e de um novo processo, possivelmente um *malware*.

No trecho de código apresentado em 3.3 é possível ver um trecho das ações executadas no sistema pelo *applet java*.

Trecho de Código 3.3. Trecho de atividade realizada pelo processo *applet Java* carregado pelo *Internet Explorer*.

```
1  javaw.exe;CreateFile;C:\WINDOWS\system32\d3d9caps.tmp
2  javaw.exe;WriteFile;C:\WINDOWS\system32\d3d9caps.tmp
```

Já o trecho de código apresentado em 3.4, que corresponde a algumas ações executadas pelo processo do *browser*, é possível ver que ele cria um novo arquivo e executa-o. Este novo processo irá realizar várias ações, sendo que as mais relevantes estão apresentadas no trecho de código 3.5. Neste pedaço de código existem trechos de execução de quatro processos: o primeiro deles corresponde às ações que vão da linha 1 a 13 e é o arquivo criado e executado pelo *Internet Explorer*, como pode ser verificado na linha 3 do trecho de código 3.4. Os outros três processos, que estão entre as linhas 15 a 21, 23 a 25 e 27 a 28, foram todos criados pelo processo `9eg1.exe`, conforme mencionado anteriormente.

Trecho de Código 3.4. Trecho de atividade realizada pelo processo do *Internet Explorer*.

```
1  iexplore.exe;CreateFile;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\9eg1.exe
2  iexplore.exe;WriteFile;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\9eg1.exe
3  iexplore.exe;CreateProcess;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\9eg1.exe
```

Trecho de Código 3.5. Trecho de atividade realizada pelos processos criados pelo *Internet Explorer*.

```
1  9eg1.exe;ConnectNet;<IP>:8000
2  9eg1.exe;SendNet;TCP:<IP>:8000
3  9eg1.exe;ReceiveNet;TCP:<IP>:8000
4  9eg1.exe;CreateFile;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\_4.tmp
5  9eg1.exe;WriteFile;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\_4.tmp
6  9eg1.exe;CreateProcess;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\_4.tmp
7  9eg1.exe;CreateFile;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\_5.tmp
8  9eg1.exe;WriteFile;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\_5.tmp
9  9eg1.exe;CreateProcess;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\_5.tmp
10 9eg1.exe;CreateFile;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\_6.tmp
11 9eg1.exe;WriteFile;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\_6.tmp
12 9eg1.exe;CreateProcess;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\_6.tmp
13 9eg1.exe;TerminateProcess;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\9eg1.ex
14
15 _4.tmp;ConnectNet;<IP>:80
16 _4.tmp;SendNet;TCP:<IP>:80
17 _4.tmp;DisconnectNet;TCP:<IP>:80
18 _4.tmp;CreateFile;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\DAT7.tmp.exe
19 _4.tmp;WriteFile;C:\DOCUME~1\ADMINI~1\LOCALS~1\Temp\DAT7.tmp.exe
20 _4.tmp;CreateFile;C:\WINDOWS\system32\drivers\str.sys
21 _4.tmp;WriteFile;C:\WINDOWS\system32\drivers\str.sys
22
```

```
23  _5.tmp;CreateFile;C:\Documents and Settings\Administrator\Local
        Settings\Temporary Internet Files\Content.IE5\ZMD946CA\gr[1].htm
24  _5.tmp;DeleteFile;C:\Documents and Settings\Administrator\Local
        Settings\Temporary Internet Files\Content.IE5\ZMD946CA\gr[1].htm
25
26  _6.tmp;CreateFile;C:\WINDOWS\system32\dll.dll
27  _6.tmp;WriteFile;C:\WINDOWS\system32\dll.dll
```

Analisando todas as linhas que mostram o comportamento dos processos criados por `9eg1.exe` é possível verificar que existem ações de conexão de rede (linhas 15 a 17), criação de *drivers* (linha 20), *DLLs* (linha 27) e criação e remoção de arquivos (linhas 23 e 24).

Uma outra informação importante obtida durante a análise é o tráfego de rede gerado pela execução do código malicioso. A partir dele é possível observar que o *malware* faz conexões em redes *P2P* utilizando o protocolo *Gnutella*. No conjunto de ações presentes no trecho de código 3.6 é possível ver a ação de *download* de uma lista de *servents* (clientes) *Gnutella* presentes na rede (linha 1) e a tentativa de conexão a um destes *servents* (linhas 16 a 22).

Trecho de Código 3.6. Requisições à rede *Gnutella* realizadas pelo *malware*.

```
1  GET /skulls.php?net=gnutella2&get=1&client=RAZA2.5.0.0 HTTP/1.1
2  Host: gwc2.wodi.org
3  Content-Type: text/html
4  Accept-Language: en
5  User-Agent: Shareaza
6  Connection: close
7  HTTP/1.1 200 OK
8  Date: Wed, 30 Mar 2011 13:28:46 GMT
9  Server: Apache/2.2.15 (Linux/SUSE)
10 X-Powered-By: PHP/5.3.3
11 Connection: close
12 X-Remote-IP: <IP>
13 Content-Length: 1257
14 Content-Type: text/plain
15 <LISTA DE HOSTs>
16 GNUTELLA CONNECT/0.6
17 Listen-IP: 0.0.0.0:18509
18 Remote-IP: <IP>
19 User-Agent: Shareaza 2.5.0.0
20 Accept: application/x-gnutella2
21 X-Ultrapeer: False
22 X-Ultrapeer-Needed: True
```

Neste capítulo foi exibida uma análise de um comprometimento que ocorreu a partir do acesso a uma *URL* maliciosa, tendo início na exploração de uma vulnerabilidade na máquina virtual Java utilizada pelo *browser* e culminando em ações nocivas efetuadas diretamente no sistema operacional. O endereço da *URL* foi obtido de um domínio que disponibiliza diariamente *links* de *sites* maliciosos. Na análise foi apresentado todo o processo do ataque, desde as ações maliciosas realizadas pelo código *HTML* da página maliciosa, até as modificações realizadas no sistema por um *malware* que foi obtido por *download*.

Referências

[1] Thorsten Holz, Markus Engelberth, Felix Freiling. Learning More About the Underground Economy: A Case-Study of Keyloggers and Dropzones. Reihe informatik tr-2008-006, University of Mannheim, 2008.

[2] B. Stone-Gross, M. Cova, B. Gilbert, R. Kemmerer, C. Kruegel, and G. Vigna. Analysis of a Botnet Takeover. *IEEE Security and Privacy Magazine*, 9(1):64–72, January 2011.

[3] Chao Yang, Robert Harkreader, and Guofei Gu. Die free or live hard? empirical evaluation and new design for fighting evolving twitter spammers. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID'11))*, September 2011.

[4] Kevin Zhijie Chen, Guofei Gu, Jose Nazario, Xinhui Han, and Jianwei Zhuge. WebPatrol: Automated collection and replay of web-based malware scenarios. In *Proceedings of the 2011 ACM Symposium on Information, Computer, and Communication Security (ASIACCS'11)*, March 2011.

[5] Michael Becher, Felix C. Freiling, Johannes Hoffmann, Thorsten Holz, Sebastian Uellenbeck, and Christopher Wolf. Mobile security catching up? revealing the nuts and bolts of the security of mobile devices. In *Proceedings of the 2011 IEEE Security and Privacy Symposium*, pages 96–111, May 2011.

[6] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2011.

[7] G. Stringhini, C. Kruegel, and G. Vigna. Detecting spammers on social networks. In *Annual Computer Security Applications Conference*, 2010.

[8] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, Newso James, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS '08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] Chia Yuan Cho, Juan Caballero, Chris Grier, Vern Paxson, and Dawn Song. Insights from the inside: a view of botnet management from infiltration. In *Proceedings of the 3rd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*, LEET'10, pages 2–2, Berkeley, CA, USA, 2010. USENIX Association.

[10] Junjie Zhang, Xiapu Luo, Roberto Perdisci, Guofei Gu, Wenke Lee, and Nick Feamster. Boosting the scalability of botnet detection using adaptive traffic sampling. In *Proceedings of the 2011 ACM Symposium on Information, Computer, and Communication Security (ASIACCS'11)*, March 2011.

[11] Seungwon Shin, Raymond Lin, and Guofei Gu. Cross-analysis of botnet victims: New insights and implications. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection (RAID'11))*, September 2011.

[12] Jason Franklin, Vern Paxson, Adrian Perrig, Stefan Savage. An inquiry into the nature and causes of the wealth of internet miscreants. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 375–388. ACM, 2007.

[13] Hamad Binsalleeh, Thomas Ormerod, Amine Boukhtouta, Prosenjit Sinha, Amr Youssef, Mourad Debbabi, Lingyu Wang. On the Analysis of the Zeus Botnet Crimeware Toolkit. In *Proceedings of the Eighth Annual Conference on Privacy, Security and Trust*, PST '2010. IEEE Press, August 2010.

[14] Seungwon Shin, Guofei Gu. Conficker and beyond: a large-scale empirical study. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 151–160. ACM, 2010.

[15] Nicholas Falliere, Liam O. Murchu, Eric Chien. Symantec Stuxnet Report: W32.Stuxnet Dossier. Report, Symantec, October 2010.

[16] Julio Canto, Marc Dacier, Engin Kirda, Corrado Leita. Large scale malware collection: lessons learned. In *27th International Symposium on Reliable Distributed Systems*, SRDS 2008. IEEE, October 2008.

[17] André R. A. Grégio, Isabela L. Oliveira, Rafael D. C. dos Santos, Adriano M. Cansian, Paulo L. de Geus. Malware distributed collection and pre-classification system using honeypot technology. In *Data Mining, Intrusion Detection, Information Security and Assurance, and Data Networks Security 2009*, volume 7344. SPIE.

[18] André R. A. Grégio, Dario S. Fernandes Filho, Vitor M. Afonso, Rafael D. C. dos Santos, Mario Jino and Paulo L. de Geus. Behavioral analysis of malicious code through network traffic and system call monitoring. In *Defense, Security and Sensing 2011*, volume 8059. SPIE.

[19] Paul Baecher, Thorsten Holz, Markus KÃ¶tter, Georg Wicherski. The Malware Collection Tool (mwcollect). PÃ¡gina na internet, 2011. `http://www.mwcollect.org/`.

[20] Joanna Rutkowska. Introducing Stealth Malware Taxonomy. White paper, 2006. `http://invisiblethings.org/papers/malware-taxonomy.pdf`.

[21] Nicholas Weaver, Vern Paxson, Stuart Staniford, and Robert Cunningham. A taxonomy of computer worms. In *Proceedings of the 2003 ACM workshop on Rapid malcode*, WORM '03, pages 11–18, New York, NY, USA, 2003. ACM.

[22] David Dagon, Guofei Gu, Cliff Zou, Julian Grizzard, Sanjeev Dwivedi, Wenke Lee, and Richard Lipton. R.: A taxonomy of botnets. In *In: Proceedings of CAIDA DNS-OARC Workshop*, 2005.

[23] Jonathon Giffin, Somesh Jha, and Barton Miller. Automated discovery of mimicry attacks. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2006.

[24] Gregoire Jacob, Eric Filiol, and Herve Debar. Functional polymorphic engines: formalisation, Implementation and use cases. *Journal in Computer Virology*, 5(3), 2008.

[25] Justin Seitz. *Gray Hat Python: Python Programming for Hackers and Reverse Engineers*. No Starch Press, San Francisco, CA, USA, 2009.

[26] Hispasec Sistemas. Virustotal. `http://www.virustotal.com/`, 2011.

[27] Nicolas Fallieri, Liam O. Murchu, and Eric Chien. W32.stuxnet dossier. `http://www.symantec.com/en/ca/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf`, 2011.

[28] Norman Sandbox. Norman sandbox whitepaper. `http://download.norman.no/whitepapers/whitepaper_Norman_SandBox.pdf`, 2003.

[29] Joris Kinable and Orestis Kostakis. Malware classification based on call graph clustering. *CoRR*, abs/1008.4365, 2010.

[30] Madhu Shankarapani, Subbu Ramamoorthy, Ram Movva, and Srinivas Mukkamala. Malware detection using assembly and api call sequences. *Journal in Computer Virology*, 7:107–119, 2011. 10.1007/s11416-010-0141-5.

[31] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA ’08, pages 108–125, Berlin, Heidelberg, 2008. Springer-Verlag.

[32] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauscheck, Christopher Kruegel, and Engin Kirda. Scalable, Behavior-Based Malware Clustering. In *16th Symposium on Network and Distributed System Security (NDSS)*, 2009.

[33] Gregoire Jacob, Matthias Neugschwandtner, Paolo Milani Comparetti, Christopher Kruegel, and Giovanni Vigna. A static, packer-agnostic filter to detect similar malware samples. Technical Report 2010-26, UCSB, November 2010.

[34] Younghee Park, Douglas Reeves, Vikram Mulukutla, and Balaji Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW ’10, pages 45:1–45:4, New York, NY, USA, 2010. ACM.

[35] Michael Bailey, Jon Oberheide, Jon Andersen, Zhuoqing Morley Mao, Farnam Jahanian, and Jose Nazario. Automated Classification and Analysis of Internet Malware. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID ’07)*, pages 178–197, Gold Coast, Australia, September 2007.

[36] Qinghua Zhang and D.S. Reeves. Metaaware: Identifying metamorphic malware. In *Computer Security Applications Conference. ACSAC 2007.*, pages 411 –420, dec. 2007.

[37] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 421 –430, dec. 2007.

[38] Gnu diff. `http://en.wikipedia.org/wiki/Diff`, 2011.

[39] The jaccard index. `http://en.wikipedia.org/wiki/Jaccard{\textunderscore}index`, 2011.

[40] Clam antivirus. `http://www.clamav.net`, 2011.

[41] Peng Li, Limin Liu, Debin Gao, and Michael K. Reiter. On challenges in evaluating malware clustering. In *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection*, RAID'10, pages 238–255, Berlin, Heidelberg, 2010. Springer-Verlag.

[42] C. Seifert, R. Steenson, I. Welch, P. Komisarczuk, and B. Endicott-Popovsky. Capture-a behavioral analysis tool for applications and documents. *digital investigation*, 4:23–30, 2007.

[43] Virtualbox, Julho 2011. `http://www.virtualbox.org/`.

[44] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing CPU emulators. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 261–272, New York, NY, USA, 2009. ACM.

[45] Galen Hunt and Doug Brubacher. Detours: binary interception of Win32 functions. In *Proceedings of the 3rd conference on USENIX Windows NT Symposium - Volume 3*, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association.

[46] Gábor Pék, Boldizsár Bencsáth, and Levente Buttyán. nEther: in-guest detection of out-of-the-guest malware analyzers. In *Proceedings of the Fourth European Workshop on System Security*, EUROSEC '11, pages 3:1–3:6, New York, NY, USA, 2011. ACM.

[47] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.

[48] Tom Liston and Ed Skoudis. On the Cutting Edge: Thwarting Virtual Machine Detection., 2006. `http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf`.

[49] Danny Quist and Val Smith. Detecting the Presence of Virtual Machines Using the Local Data Table, 2006. http://www.offensivecomputing.net/files/active/0/vm.pdf.

[50] Mendel Rosenblum. The Reincarnation of Virtual Machines. *Queue*, 2:34–40, July 2004.

[51] Microsoft Portable Executable and Common Object File Format Specification. http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx.

[52] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Microsoft Press, Redmond, WA, USA, 2004.

[53] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy*, 5(2):32–39, March 2007.

[54] CWSandbox :: Behavior-based Malware Analysis, Julho 2011. http://mwanalysis.org/.

[55] Cuckoo Sandbox - Automated Malware Analysis System, MarÃ§o 2011. http://www.cuckoobox.org/.

[56] JoeBox, Julho 2011. http://www.joesecurity.org/.

[57] Anubis - Analyzing Unknown Binaries, MarÃ§o 2011. http://anubis.iseclab.org/.

[58] CaptureBat, MarÃ§o 2011. http://www.honeynet.org/project/CaptureBAT.

[59] Greg Hoglund and Jamie Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[60] Thomas Raffetseder, Christopher Krugel, and Engin Kirda. Detecting System Emulators. In *ISC*, pages 1–18, 2007.

[61] Bill Blunden. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Jones and Bartlett Publishers, Inc., USA, 2009.

[62] Holy Father. Hooking Windows API-Technics of Hooking API Functions on Windows, 2004.

[63] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[64] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. TTanalyze: A Tool for Analyzing Malware, 2006.

[65] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 281–290, New York, NY, USA, 2010. ACM.

[66] J. Nazario. Phoneyc: A virtual client honeypot. In *Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*, pages 6–6. USENIX Association, 2009.

[67] C. Seifert and R. Steenson. Capture - honeypot client (capture-hpc), 2006.

[68] Vmware, Julho 2011. `http://www.vmware.com/`.

[69] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating emulation-resistant malware. In *Proceedings of the 1st ACM workshop on Virtual machine security*, VMSec '09, pages 11–22, New York, NY, USA, 2009. ACM.