Análise Comportamental de Código Malicioso Através da Monitoração de Chamadas de Sistema e Tráfego de Rede

Dario S. Fernandes Filho[1,2], André R. A. Grégio[1,2], Vitor M. Afonso[1,2], Rafael D. C. Santos[3], Mário Jino[1], Paulo L. de Geus[1]

[1]Universidade Estadual de Campinas (Unicamp) - Campinas - SP - Brasil

[2]Centro de Tecnologia da Informação Renato Archer (CTI) - Campinas - SP - Brasil

[3]Instituto Nacional de Pesquisas Espaciais (INPE) - São José dos Campos - SP - Brasil

{dario, vitor, paulo}@las.ic.unicamp.br, argregio@cti.gov.br, jino@dca.fee.unicamp.br, rafael.santos@lac.inpe.br

Abstract. *Malicious code (malware) spread through the Internet—such as viruses, worms and trojans—is a major threat to information security nowadays and a profitable business for criminals. There are several approaches to analyze malware by monitoring its actions while it is running in a controlled environment, which helps to identify malicious behaviors. In this article we propose a tool to analyze malware behavior in a non-intrusive and effective way that extends the analysis possibilities to cover malware samples that bypass current approaches and also fixes some issues with them, filling a gap in the field.*

Resumo. *Código malicioso (malware) disseminado através da Internet—vírus, worms, trojans—é a maior ameaça atual à segurança da informação e um negócio lucrativo para criminosos. Há abordagens para analisar malware que monitoram suas ações durante a execução em ambiente controlado, permitindo identificar comportamentos maliciosos. Neste artigo, propõe-se uma ferramenta de análise comportamental de malware não intrusiva, a qual amplia a análise a exemplares que contornam as abordagens atuais e corrige alguns problemas presentes nestas, preenchendo assim uma lacuna na área.*

1. Introdução

A disseminação de código malicioso através da Internet tem sido a maior ameaça atual à segurança de sistemas de informação. Por código malicioso, ou *malware*, entende-se o conjunto de aplicações também denominadas individualmente como vírus, *worms*, *keyloggers*, *trojans*, *backdoors* e outros tipos de programas com a intenção de comprometer um sistema. A crescente interação de dispositivos diversos com a Internet, aliada à vasta provisão de serviços e falta de conhecimento adequado por parte dos usuários contribui para o cenário atual de ataques por meio de *malware*. A motivação por trás desses ataques é a economia que foi estabelecida baseada em aluguel de infraestruturas comprometidas e venda de informações sensíveis, tais como contas e senhas bancárias e números de cartão de crédito [Holz et al. 2008] [Franklin et al. 2007].

Embora de tempos em tempos certa quantidade de grupos de *malware* esteja em atividade—às vezes causando sérios incidentes de proporções globais, tais como os recentes resultados do alastramento dos *malware Zeus* [Binsalleeh et al. 2010] e *Conficker* [Leder e Werner 2009]—as atividades realizadas no sistema vítima sempre podem ser mapeadas para ações específicas, pois envolvem desabilitar mecanismos de

proteção (*firewall*, antivírus), modificar binários e bibliotecas do sistema operacional, alterar chaves do registro, abrir portas para comunicação, entre outras.

Assim, um dos métodos que podem ser utilizados para a identificação de *malware* é o levantamento de suas ações no sistema vítima, por meio da análise do código malicioso. A análise de *malware* pode ser dividida em dois tipos: análise estática e análise dinâmica. Na análise estática, informações sobre o código do *malware* são obtidas sem que seja necessário executá-lo. Já na análise dinâmica, o comportamento do *malware* é monitorado durante sua execução. A realização da análise de *malware* pode auxiliar tanto na automatização de parte do processo de análise feito pelas empresas de antivírus, quanto no fornecimento de informações que podem ser úteis na criação de novos meios de identificação e remediação.

Devido ao problema da ofuscação de *malware*, que leva a diferentes representações do programa para gerar o mesmo resultado, a análise estática torna-se difícil no que diz respeito à obtenção de resultados confiáveis. Essa ofuscação é, em parte, ineficiente no caso da análise dinâmica. Existem alguns sistemas de análise dinâmica de *malware* cuja utilização é disponibilizada através da Internet, cada qual diferenciado por suas peculiaridades técnicas. Dentre tais sistemas, os que mais se destacam são o Anubis [Bayer et al. 2006], CWSandBox [Willems et al. 2007], BitBlaze [Song et al. 2008], Ether [Dinaburg et al. 2008] e JoeBox [JoeBox 2010], que executam o *malware* em sistema operacional Windows XP e monitoram suas atividades por um período limitado de tempo, gerando relatório da análise.

Todas as abordagens de análise possuem deficiências, sendo que a principal delas é sucumbir a mecanismos de detecção presentes nos *malware*, os quais identificam que estão sendo monitorados e terminam sua execução ou modificam seu comportamento. Neste trabalho é apresentado o sistema BehEMOT (*Behavior Evaluation from Malware Observation Tool*), que analisa *malware* dinamicamente de forma automatizada, com a finalidade de levantar as ações realizadas no sistema vítima e prover meios para identificação de comportamentos maliciosos, além de suprir algumas das deficiências apresentadas por outras abordagens de análise dinâmica existentes. As principais contribuições dadas por BehEMOT são:

- A arquitetura de BehEMOT foi projetada para a utilização mista de ambiente de análise emulado e real, contornando determinadas técnicas usadas em alguns *malware* para detecção de máquinas virtuais e detecção de emuladores;
- A monitoração de chamadas de sistema (*system calls*) nativas do *kernel* do Windows utilizada simplifica o processo de levantamento de comportamentos, obtendo tanto as funções quanto seus argumentos;
- O método de monitoração implementado por BehEMOT possibilita que, em uma única análise, todos os processos-filho criados pelo *malware* também sejam monitorados, além de garantir que não haja ruído referente às chamadas de sistema de outros processos não associados ao objeto de análise;
- A execução do *malware* através do BehEMOT não utiliza técnicas intrusivas, evitando assim que o *malware* perceba que está sendo monitorado e altere sua execução;

- As atividades de rede são também capturadas fora do ambiente de análise, permitindo inclusive a obtenção de informações acerca do comportamento de *rootkits* que porventura se comuniquem através da interface de rede.

O restante do artigo está dividido como descrito a seguir. Na Seção 2, são apresentados os conceitos de análise estática e dinâmica de *malware*, as vantagens e desvantagens de cada tipo de análise e algumas técnicas de análise estática. Na Seção 3, são apresentadas as principais abordagens existentes para análise dinâmica. A Seção 4 contém o detalhamento da arquitetura do BehEMOT e da implementação de seus componentes. Na Seção 5 são mostrados os testes realizados para validar o sistema e os resultados de análise obtidos. Na Seção 6 são feitas as considerações finais sobre o sistema, uma breve comparação com outros sistemas existentes e os trabalhos futuros que podem ser gerados.

2. Análise de Código Malicioso

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.

Por outro lado, a análise dinâmica pode ser comprometida por técnicas anti-análise utilizadas por certos tipos de *malware* em tempo de execução, bem como permite apenas que se observe um dos caminhos que seu código pode escolher. A seguir, é apresentada uma visão geral do processo de análise estática.

2.1. Análise Estática

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. Técnicas para Análise Dinâmica Automatizada

Nesta Seção, são descritas as principais abordagens para análise dinâmica automatizada de código malicioso, as quais são empregadas nos principais sistemas de análise de *malware* disponíveis. Tais sistemas têm por foco analisar binários do tipo PE-32 *executable* em sistema operacional Windows, o qual é um formato de arquivos do Windows que define executáveis e *Dynamic Link Libraries* (DLL)—bibliotecas que contêm implementações de funções. Neste tipo de arquivo estão presentes as instruções *assembly* que representam o que deve ser feito no sistema durante sua execução, além de definições necessárias para sua inicialização [Microsoft PECOFF 2008].

As ações realizadas pelo binário compreendem certas atividades efetuadas no sistema operacional, tais como abrir um arquivo, realizar alterações em registros, abrir portas de comunicação de rede, criar processos, etc. Essas atividades denotam o comportamento de um arquivo executável. Assim, pode-se definir o comportamento de um *malware* como sendo o conjunto ordenado das atividades realizadas por este no sistema e como o sistema reage às tais atividades. Para obter um comportamento de *malware* é preciso executá-lo em um sistema especialmente projetado para a tarefa de monitoração. Com esta finalidade, existem técnicas específicas, sendo que as principais são *Virtual Machine Introspection*, *System Service Dispatch Table hooking* e *Application Programming Interface hooking*, as quais serão explicadas a seguir.

Virtual Machine Introspection (VMI) [Garfinkel e Rosenblum 2003] é um tipo de análise que utiliza um ambiente virtual (*guest system*) para a execução do *malware* e uma camada intermediária entre o ambiente virtual e o real (*host system*), chamada *Virtual Machine Monitor* (VMM). Esta camada intermediária é responsável por obter e controlar as ações que estão sendo executadas no ambiente virtual, isolando-o e minimizando a chance de comprometimento do ambiente real do sistema de análise. Com esta técnica é possível obter informações de mais baixo nível sobre a execução do binário, como por exemplo, as chamadas de sistema (*system calls*) executadas e o estado da memória e dos registradores do processador. Porém, a grande limitação desta abordagem é justamente a necessidade do ambiente de análise virtualizado. Existem

alguns tipos de *malware* que, a fim de burlar a análise e identificação de suas ações, realizam diversas checagens para verificar se estão executando em ambiente virtual e que, ao notarem a presença deste tipo de ambiente, modificam seu comportamento de forma a ocultar o fluxo de execução malicioso [Balzarotti et al. 2010].

A técnica de *System Service Dispatch Table (SSDT) hooking*, ou captura de chamadas do sistema trata da interceptação das chamadas no sistema operacional através de funções de *hook*, possibilitando inclusive que se modifique o fluxo de execução [Kong 2007] e as respostas retornadas ao programa [Hoglund e Butler 2005]. Esta é aplicada através do uso de um *driver*, isto é, um programa especial que executa no nível do *kernel* (*ring 0* nos sistemas Windows) e normalmente é utilizado para fazer a interface entre dispositivos de *hardware* e o nível de usuário (*ring 3*). A operação em *ring 0* e conseqüente posse de maiores privilégios de acesso possibilita que um *driver* intercepte todas as chamadas de sistema realizadas por um determinado programa que executa em nível de usuário. Esta técnica pode ser usada tanto em sistema real quanto em virtual. Isso ocorre devido ao fato de que, para se realizar a monitoração do *malware* durante a análise basta que se instale o *driver* no sistema monitorado. Outra vantagem do uso de *SSDT hooking* é que não se modifica o código do arquivo monitorado, evitando que o *malware* descubra que está sendo analisado através de checagem de integridade. A desvantagem da aplicação desta técnica diz respeito à análise de alguns tipos de *rootkits*, uma classe particular de *malware* que executa em nível de *kernel* e geralmente é implementada sob a forma de um *driver*. Neste caso, como o malware estará executando no mesmo nível de privilégio do *driver* de monitoração, ele poderá executar ações que não serão capturadas.

Outra técnica utilizada é a de *hooking* de APIs, na qual a captura das informações é feita através de modificações no código do binário em análise, de modo que os endereços das APIs que o programa vai executar sejam trocados por endereços de funções do programa responsável pela interceptação dos dados. No momento em que o *malware* é carregado no sistema, todas as *DLL*s utilizadas são verificadas para que sejam obtidos os endereços das APIs que se quer modificar [Father 2004]. Após este levantamento, a troca dos endereços é feita. Esta técnica pode ser facilmente subvertida se o *malware* realizar chamadas diretas ao *kernel* do sistema. Neste caso, como não há referência a nenhuma API, o programa de monitoração não é capaz de capturar as ações efetuadas, fazendo com que se perca um subconjunto crítico das atividades do comportamento deste *malware*. Outra desvantagem da utilização desta técnica é que o *malware* pode identificar facilmente que está sendo analisado através de checagem de integridade, isto é, verificando se houve modificações no código em execução.

A partir do estudo das características destas técnicas, escolheu-se a de *SSDT hooking* para implementação da ferramenta proposta neste artigo, por não necessitar de ambiente virtualizado e possibilitar em um só passo a obtenção das chamadas de sistema referentes ao processo monitorado sem modificação de seu código.

4. Detalhamento do Sistema

Nesta Seção é descrita a arquitetura do sistema correspondente à ferramenta proposta neste artigo, detalhando-se seus componentes internos. Conforme apresentado, BehEMOT é um sistema de análise comportamental de *malware* cujo foco são os

arquivos binários executáveis em sistemas operacionais Windows. O fluxo de dados pode ser resumido da seguinte forma:

- Um exemplar de *malware* é inserido no sistema através de um *script* de controle, o qual verifica se o *malware* está presente no banco de dados de exemplares e se já foi analisado;
- O *malware* é executado em um ambiente Windows XP SP3 instalado de forma padrão, com a adição da ferramenta BehEMOT—*driver* para aplicação de *SSDT hooking* e seu controlador;
- A execução inicial é limitada por um período de quatro minutos e o ambiente Windows XP é emulado através da ferramenta Qemu [Bellard 2005];
- O tráfego de rede é capturado no mesmo sistema onde se encontra o *script* de controle, isto é, em um sistema externo ao da análise;
- Caso não sejam capturadas atividades referentes à execução deste *malware*, ou a execução termine com erro, este é submetido automaticamente a um ambiente similar em máquina real (sem emulação ou virtualização);
- Após a análise, o *script* de controle recebe o comportamento do *malware* e o tráfego de rede capturado correspondente, insere-os no banco de dados e processa as informações para gerar um relatório da análise comportamental.

Nas próximas Subseções são explicados a arquitetura do sistema como um todo, o *driver*, seu controlador e o processo de análise comportamental.

4.1. Arquitetura e Componentes

A ferramenta BehEMOT foi projetada de forma modular, sendo composta do *driver* que implementa *SSDT hooking* (detalhado na Subseção 4.2), do controlador deste *driver* (Subseção 4.3) e do analisador (Subseção 4.4), o qual interage com o ambiente de análise comportamental com a finalidade de obter as ações do *malware* e extrair o comportamento. Tal arquitetura pode ser vista na Figura 1. O ambiente completo é formado por um *pool* de sistemas operacionais emulados através do Qemu, um sistema operacional real (não emulado ou virtualizado), um *firewall* para captura do tráfego de rede, isolamento dos sistemas e contenção de ataques lançados e um banco de dados para armazenamento de *malware* e informações de análise.

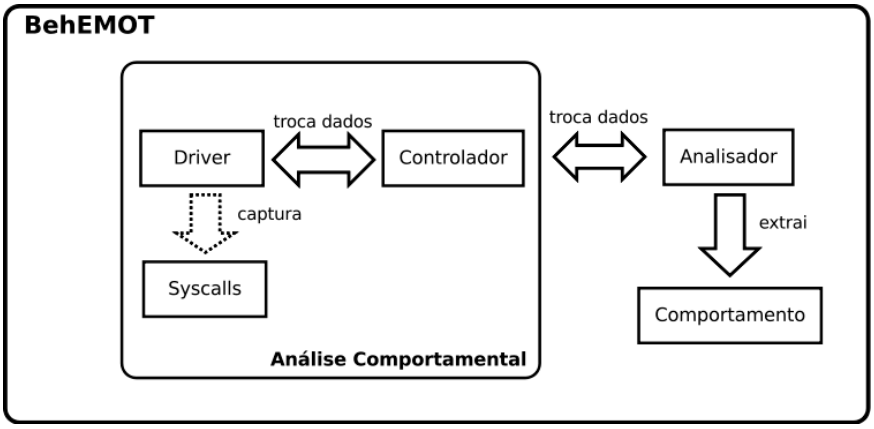

Figura 1. BehEMOT em componentes.

Tanto o ambiente real quanto o emulado contêm as mesmas configurações e possuem o Controlador e o *Driver* que compõem BehEMOT, os quais são responsáveis por capturar as chamadas de sistema realizadas pelo *malware* e seus processos-filhos, bem como por enviar esses dados para o Analisador, situado fora do ambiente de monitoração (análise comportamental). Além disso, os ambientes têm instalado como sistema operacional um Windows XP com *Service Pack* 3, de forma padrão. Não são inseridas aplicações ou ferramentas adicionais, além das citadas acima, nem são feitas configurações especiais de qualquer tipo, tornando o ambiente similar ao de grande parte dos usuários médios que utilizam tal sistema operacional. Como explicitado, um dos ambientes foi criado através do Qemu, um emulador de PC *open-source*, no qual são emulados todos os dispositivos do sistema, bem como o processador onde serão executadas as instruções *assembly* do *malware*. Este ambiente é executado na mesma máquina do Analisador como um processo independente, e a comunicação é feita por meio de uma interface de rede virtual.

Com a utilização deste ambiente, é possível fazer a análise comportamental de forma eficiente através do uso de *snapshots*, funcionalidade provida pelo Qemu na qual o sistema é iniciado a partir de um estado previamente salvo com os mecanismos de monitoração do BehEMOT já instanciados, fazendo com que o tempo de inicialização do sistema operacional seja desconsiderado. Outra vantagem é a facilidade na restauração do sistema ao seu estado íntegro após o comprometimento causado pela execução do *malware*.

Já no ambiente real de análise, foi gerada uma imagem do sistema operacional íntegro em uma partição separada, a qual é manuseada pela ferramenta *Partimage*, que salva e restaura estados de uma dada partição. Esse processo de restauração é feito por um sistema operacional Linux, instalado e carregado em outra partição da máquina deste ambiente. Esta partição é ativada sempre que se completa uma análise comportamental, restaurando o Windows ao estado anterior à execução do *malware*. A comunicação entre o Analisador e o Controlador no ambiente real é feita através da interface de rede, em uma subrede local.

4.2. Driver

O *Driver* é o componente principal do sistema, operando em *ring 0*. Este componente é responsável por capturar as chamadas de sistema executadas por um determinado processo, através da interceptação da *System Service Dispatch Table* (*SSDT hooking*).

Certas estruturas no kernel do *Windows* armazenam informações referentes ao funcionamento do sistema operacional—processos e *threads*, por exemplo. Uma dessas estruturas é a SSDT, uma tabela onde ficam os endereços de memória das APIs nativas do sistema [Blunden 2009]. A SSDT consiste basicamente de um vetor de endereços de memória, onde cada índice corresponde a uma chamada de sistema. Sempre que uma chamada de sistema nativa precisa ser executada, esta tabela é consultada para se obter o endereço da função apropriada.

A obtenção destes endereços é feita através do acesso a uma estrutura exportada pelo kernel chamada `KeServiceDescriptorTable`. Tal estrutura só pode ser manipulada por aplicações que executam em nível de kernel e é composta por 4 campos, cada qual armazenando uma informação específica. Os campos relevantes para

a realização do *hooking* são o primeiro, que contém um apontador para o início da SSDT, e o terceiro, onde o número de itens presentes na SSDT é armazenado.

A técnica de SSDT *hooking* consiste da alteração dos endereços contidos na SSDT por novos endereços cujas funções se tem controle. No processo de inicialização do *Driver*, o procedimento para realizar o *hooking* começa com a desativação da proteção que impede a escrita na memória, especificamente na `KeServiceDescriptorTable`. Para tal, esta tabela é mapeada através da *Memory Descriptor List* (MDL), uma estrutura que descreve uma dada região de memória e permite que a escrita seja habilitada, possibilitando assim a substituição dos endereços contidos na SSDT.

A cada endereço modificado da SSDT, faz-se necessária a associação de uma função, a qual aponta para a chamada de sistema original para que seja possível interceptar ou modificar os resultados obtidos, além de manter o funcionamento normal do sistema. O endereço original contido na SSDT deve ser armazenado antes de realizar-se a substituição, para que possa ser utilizado posteriormente. A troca das funções (nova e original) é feita atomicamente pelo *Driver*.

Com o término do processo de inicialização do *Driver* o sistema estará modificado, de forma que cada chamada de sistema realizada irá passar pelas funções definidas no *Driver*. Dado que existem 391 chamadas de sistema na SSDT [Blunden 2009], foram escolhidas aquelas que apresentam informação relevantes sobre a execução do artefato [Bayer et al. 2009] e que podem ser úteis para denotar um comportamento malicioso. Estas chamadas são referentes a operações de modificação de arquivos, modificação de chaves de registro, operações envolvendo *mutexes* e operações de processos e *threads*.

Para cada uma das chamadas de sistema selecionadas foi criada uma nova função que realiza a chamada original e armazena os parâmetros utilizados, caso o processo que a está executando esteja marcado para monitoração. A escolha das chamadas de sistema que devem ser monitoradas é baseada no *Process Identifier* (PID) do processo que a invoca. Os processos que devem ser monitorados são definidos pelo Controlador e pelas chamadas de sistema de criação dos processos-filhos. O Controlador envia para o *Driver* através de *I/O Request Packet* (IRP) o PID do processo do *malware* que deve ser analisado. Se, durante a execução, o *malware* criar algum novo processo, o PID deste também será marcado para monitoração. Caso o processo seja finalizado, seu PID é removido e a monitoração deste cessa.

Além dos parâmetros, a origem do executor da ação também é armazenada, facilitando assim a análise posterior. As ações são representadas por linhas cujos campos são divididos em *timestamp*, executor, tipo da operação e alvo, exemplificadas na Tabela 1. As informações coletadas são periodicamente lidas pelo Controlador via IRP, e então são enviadas para o Analisador, explicados nas próximas subseções.

Tabela 1. Exemplo de ações pertencentes ao comportamento de um dado *malware*, separadas por seus campos.

Timestamp	Executor	Tipo da operação	Alvo
13:58:48.402	malware.exe	CREATE MUTEX	_AVIRA_21099
13:58:48.480	malware.exe	WRITE FILE	C:\WINDOWS\system32\sdra64.exe
13:58:48.965	malware.exe	OPEN PROCESS	C:\WINDOWS\system32\winlogon.exe

4.3. Controlador

A interface entre os dados coletados pelo *Driver* e o ambiente externo ao da análise é feita através do Controlador, um programa que executa na área do usuário (*Ring 3*). O Controlador é o responsável por criar o processo do *malware* que deve ser analisado, além de inicializar e finalizar o *Driver*, bem como controlar e interpretar as informações fornecidas por este durante a execução. Após a inicialização do Controlador, este irá inicializar o *Driver* utilizando o *Service Control Manager* (SCM). Isto é necessário para que o *Driver* não seja paginado durante a execução, evitando erros não tratados pelo *kernel* (BSOD—*Blue Screen of Death*).

Ao se completar a etapa inicial, o processo do *malware* é criado em estado suspenso de modo que seja possível obter o seu PID, o qual é repassado para o *Driver*. Com isto, identifica-se o processo que deve ser monitorado e seu estado é retomado, iniciando a execução do *malware*. Deste ponto em diante, o Controlador permanece em estado de verificação para capturar os dados enviados pelo *Driver*, que são formatados e armazenados em um arquivo. Por fim, após um *timeout*, o arquivo com as chamadas de sistema é fechado e o Controlador o envia para o Analisador.

4.4. Analisador

O Analisador é o componente responsável por encaminhar o processo da análise dinâmica, enviando um *malware* para o ambiente de análise e obtendo o resultado das chamadas de sistema efetuadas. É ele quem decide se o *malware* a ser analisado deve ser enviado para a máquina real ou para o sistema emulado, baseado em dois princípios:

-Se o *malware* em questão possui *packer* e este é identificado como algum dos que causam problemas de execução em ambientes usando Qemu, tais como *TE!Lock*, *Armadillo* e certas versões do *PECompact* [Balzarotti et al. 2010];

- Se o *malware* já foi executado no ambiente emulado e a execução terminou com erro, o que pode acontecer em caso de detecção do ambiente.

Após o término da análise dinâmica, é papel do Analisador tratar o arquivo com as chamadas de sistema a fim de gerar o relatório de comportamento do *malware*. Neste relatório, além das informações extraídas da interação com o sistema operacional, são agregadas as atividades de rede que o *malware* tentou realizar, como *download* ou envio de arquivos, varreduras, requisições de DNS ou ataques.

5. Testes e Resultados

A avaliação do sistema foi feita através da execução de algumas amostras de *malware* selecionadas aleatoriamente do banco de dados de exemplares instanciado para armazenar os artefatos coletados no âmbito deste trabalho. A escolha primou os exemplares que possuíam representatividade quanto à identificação por mecanismo antivírus e *packing* por ferramentas que causam problemas em Qemu, como as citadas na Seção anterior.

Os objetivos deste teste foram validar o funcionamento da arquitetura do sistema de análise como um todo (cujo acesso será disponibilizado publicamente em breve) avaliar a qualidade da ferramenta na extração de comportamentos relevantes para a compreensão das atividades do *malware* no sistema vítima e verificar sua efetividade na análise de uma variedade maior de tipos de exemplares através da abordagem híbrida

(sistema operacional emulado e real). Tais verificações foram feitas de duas formas: primeiro, obtendo-se a análise publicada de um exemplar de referência e comparando-a com o resultado do BehEMOT; por último, por meio da submissão das amostras de *malware* escolhidas para o BehEMOT e para os sistemas disponíveis publicamente mais conhecidos e utilizados (Anubis e CWSandBox), e conseqüente comparação dos resultados.

Um dos exemplares de referência escolhido foi o *malware Allaple*, um *worm* polimórfico [SoftPanorama 2009] cujo comportamento predominante caracteriza-se pela cópia de seu executável em `%System%\urdvxc.exe`, modificação de chave de registro (`[HKCR\CLSID\{CLSID}\LocalServer32]`) para execução automática após a reinicialização da máquina [SecureList 2007], geração de tráfego ICMP com conteúdo característico e tentativas de acesso a diversos endereços IP com atividades de NetBIOS [SoftPanorama 2009].

A comparação do resultado da execução do exemplar de referência do *Allaple* no BehEMOT em relação ao comportamento predominante esperado pode ser observada na Tabela 2, a qual contém trechos relacionados às ações retiradas do comportamento obtido do Analisador.

Neste estudo de caso apresentado, foi obtido um comportamento composto por 302 ações no sistema até o término do processo principal (identificado como *malware.exe*) e do processo-filho criado (*urdvxc.exe*). Através deste comportamento, pôde-se observar todas as chaves de registro que foram lidas, criadas ou modificadas, os arquivos que foram criados, acessados e apagados pelos processos, bibliotecas que foram carregadas, acesso à rede, entre outras ações efetuadas, o que permitiu a correlação do exemplar analisado com um *malware* conhecido.

Tabela 2. Comportamento predominante do *malware* de referência (*worm Allaple*) e parte do comportamento obtido dinamicamente por BehEMOT.

Comportamento esperado	Comportamento obtido (BehEMOT)
%System%\urdvxc.exe	malware.exe;WRITEFILE;%System%\urdvxc.exe
HKCR\CLSID\{CLSID}\LocalServer32	urdvxc.exe;WRITEREGISTRY;HKCR\CLSID\{CLSID}\LocalServer32
TRÁFEGO ICMP	16:54:22.099043 IP BEHEMOT > **xxx.yy.aa.8**: **ICMP echo request**, id 512, seq 1792, length 41 [...] 16:54:22.223208 IP BEHEMOT > **xxx.yy.ee.23: ICMP echo request**, id 512, seq 2816, length 41
TRÁFEGO NetBIOS	16:54:27.073153 IP BEHEMOT.1060 > **xxx.yy.ww.155.netbios-ssn**: Flags [S] [...] 16:54:27.146635 IP BEHEMOT.1063 > **xxx.yy.rrr.126.netbios-ssn**: Flags [S]

De maneira complementar, fez-se testes de execução com outros exemplares de *malware*, também submetidos aos outros dois sistemas de análise dinâmica citados anteriormente e que utilizam técnicas diferentes para monitoração de atividades maliciosas. Para este teste foram selecionados exemplares de *worms*, *trojans*, vírus e *bots* já identificados por mecanismos antivírus e, em seguida, foi feita a análise dos relatórios providos pelos três sistemas em comparação, sendo um deles o proposto neste artigo. A Tabela 3 mostra os resultados obtidos deste teste, no qual se comparou o comportamento predominante que o *malware* deveria apresentar com as atividades

capturadas nos ambientes de análise, levando-se em conta a presença de tráfego de rede, comportamento relevante para identificação (ações no sistema vítima) e não interferência no funcionamento normal dos sistemas, tais como identificação de monitoração, problemas de execução no ambiente por limitação da tecnologia utilizada e presença de ruído proveniente de atividades não relacionadas à execução do *malware*.

Com este teste foi possível notar que o sistema de análise CWSandbox apresenta informações que não dizem respeito ao comportamento gerado pelo *malware*. Processos responsáveis pelo controle do sistema de análise, serviços nativos do sistema operacional e programas de atualização aparecem no relatório gerado pelo referido sistema. Essa informação pode confundir o usuário, levando-o a crer que eles foram executados pelo *malware* durante a análise. Outro problema apresentado é a facilidade na detecção do ambiente de análise, pois como foi possível identificar os programas responsáveis pelo controle da análise, basta realizar uma simples procura na execução do programa para que o ambiente seja detectado. Já no caso do sistema Anubis, houve problema com alguns exemplares que tiveram sua execução interrompida. O comportamento apresentado nessas análises estava incompleto, restringindo-se somente até a ocorrência da falha.

Tabela 3. Detecção de comportamento relevante de exemplares de *malware* analisados dinamicamente em BehEMOT, Anubis e CWSandBox, onde "Sim" indica que a monitoração do *malware* ocorreu com sucesso e os resultados continham informações úteis e "Não" significa que o *malware* não executou adequadamente no ambiente de análise, utilizou técnicas que inviabilizaram a análise e/ou havia ruído demais no relatório impossibilitando a verificação do comportamento do exemplar. Os nomes dos exemplares foram obtidos através do antivírus ClamAV e dos *packers*, quando identificados, através do PEFile [Choi et al. 2008].

Exemplar (*Packer*)	BehEMOT	Anubis	CWSandBox
Trojan.Agent (Armadillo v1.71)	Sim	*Não*	*Não*
Trojan.Agent (kkrunchy 0.23 alpha -> Ryd)	Sim	*Não*	Sim
Worm.Allaple	Sim	Sim	Sim
Worm.Padobot	Sim	Sim	Sim
Worm.Palevo	Sim	*Não*	*Não*
Trojan.Buzus	Sim	Sim	*Não*
Trojan.Ircbot (MEW 11 SE v1.2 -> Northfox[HCC])	Sim	Sim	*Não*
Trojan.Sdbot	Sim	Sim	*Não*
Trojan.Small	Sim	Sim	*Não*
W32.Virut	Sim	Sim	Sim
PUA.Packed.tElock1.Private (tElock 0.98 -> tE!)	Sim	*Não*	*Não*

6. Considerações Finais

A análise comportamental de *malware* baseada na monitoração de suas ações em um sistema operacional alvo pode gerar informações de grande utilidade relacionadas à defesa de sistemas e redes, seja visando a compreensão do funcionamento do exemplar malicioso, o desenvolvimento de *patches* para o sistema operacional e suas aplicações, a criação de assinaturas ou heurísticas para mecanismos de proteção, ou a resposta a incidentes de segurança. Assim, é de extrema importância que a ferramenta ou ambiente que propicia a análise dinâmica do *malware* proveja resultados sucintos e diretos, sem a presença de ruídos provenientes de ações não relacionadas à execução do exemplar no sistema.

Entretanto, verificou-se nos relatórios obtidos de alguns sistemas de análise disponíveis publicamente uma grande quantidade de atividades monitoradas que não pertenciam às atividades relativas à execução do *malware* (processos e aplicações executados normalmente pelo sistema operacional), bem como atividades referentes aos processos de monitoração do comportamento do *malware* e de controle do ambiente de análise. Em outros casos, pôde-se notar que não houve filtragem dos dados de saída para inserção no relatório, incorrendo em informações irrelevantes e repetitivas que não acrescentam valor à análise. BehEMOT trata tais problemas monitorando diretamente o processo malicioso e os processos-filhos criados por este.

Desta forma, o relatório gerado pelo sistema de análise BehEMOT é mais compacto e compreensível se comparado ao do Anubis, CWSandBox e JoeBox (não presente na tabela por utilizar o mesmo tipo de ambiente de monitoração do CWSandBox). O relatório do BehEMOT contém informações pontuais as quais explicitam a cadeia ordenada de ações executadas pelo *malware* e seus processos-filhos no sistema durante sua execução, além de informações referentes ao tráfego de rede gerado, caracterizando o comportamento do exemplar no sistema alvo e possibilitando, assim, a definição de uma contra-medida para casos de comprometimento com esse *malware* específico.

Um outro ponto interessante é que determinadas abordagens que se utilizam de apenas um tipo de ambiente de monitoração (emulação, virtualização ou ambiente real) padecem dos problemas específicos de detecção ou técnicas de anti-análise de cada ambiente e da limitação do método usado (emuladores não executam certas instruções, máquinas virtuais são facilmente detectadas, máquinas reais não escalam bem e apresentam problemas no *hardware* com a sobrecarga de utilização). Como BehEMOT utiliza uma abordagem mista de emulação e ambiente real, o problema é minimizado, gerando uma vazão maior do número de análises realizadas completamente.

Todas as abordagens de análise dinâmica que necessitam de um *driver* inserido no sistema operacional monitorado para captura de chamadas de sistema estão sujeitas a apresentar problemas com *rootkits*. Dependendo de como o *rootkit* é instalado, pode haver conflito com o *driver* de monitoração e ocorrer um erro no *kernel* do sistema e a não realização da análise. Este tipo de situação só pode ser contornada caso o ambiente de análise seja monitorado externamente por completo, tornando inviável a análise em máquina real. Outro problema da monitoração de *rootkits* ocorre quando há a ocultação de algum processo malicioso criado, impedindo a obtenção do PID. Nesse caso, BehEMOT é capaz de monitorar o processo inicial até o ponto da ocultação das atividades, o que pode gerar um comportamento incompleto. Por outro lado, a captura externa do tráfego de rede (em complementação às atividades capturadas pelo *driver*) pode agregar informações ao relatório que permitam a identificação de certas ações maliciosas.

Embora a abordagem mista trate alguns problemas relacionados à técnicas anti-análise, caso a quantidade de *malware* aplicando tais técnicas seja muito superior aos *malware* passíveis de execução em ambiente emulado, as mesmas desvantagens de uso intensivo de máquina real se aplicam. Além disso, a análise atual é limitada a binários executáveis em Windows XP, o que faz com que se perca aqueles que executam em outras versões de Windows ou outra plataforma, e outros tipos de aplicações que podem apresentar comportamento malicioso, como arquivos PDF, JavaScript ou *Flash*.

Por fim, embora ainda não se tenha testado exaustivamente o sistema com milhares de exemplares de *malware*, a diversidade das amostras testadas aponta para resultados promissores, sendo que o ambiente pode ser estendido para lidar com tipos diferentes de arquivos. Como trabalhos futuros estão sendo desenvolvidas as seguintes atividades: geração de árvores dos comportamentos obtidos pelo BehEMOT, pontuação de ações executadas a fim de detectar um comportamento como malicioso, análise comportamental de arquivos PDF, monitoração de atividades maliciosas efetuadas em *browsers* e geração de lista de ações para remediação do sistema comprometido.

Referências

Balzarotti, D., Cova, M., Karlberger, C., Kruegel, C., Kirda, E. and Vigna, G. (2010) "Efficient detection of split personalities in malware", 17th Annual Network and Distributed System Security Symposium, February 28th-March 3rd.

Bayer, U., Habibi, I., Balzarotti, D., Kirda, E., and Kruegel, C. (2009), "A View on Current Malware Behaviors", Usenix Workshop on Large-scale Exploits and Emergent Threats (LEET). USA, April.

Bayer, U., Kruegel, C. and Kirda, E. (2006). "TTanalyze: A Tool for Analyzing Malware", Proc. 15th Ann. Conf. European Inst. for Computer Antivirus Research (EICAR), 2006, pp. 180–192.

Bellard, F. (2005) "QEMU, a fast and portable dynamic translator", In Proceedings of the Annual Conference on USENIX Annual Technical Conference, USENIX Association, p. 41-41.

Binsalleeh, H., Ormerod, T., Boukhtouta, A., Sinha, P., Youssef, A., Debbabi, M. and Wang, L. (2010). "On the Analysis of the Zeus Botnet Crimeware Toolkit". In the Proceedings of the Eighth Annual Conference on Privacy, Security and Trust, PST'2010, Aug 17-19, Ottawa, ON, Canada, IEEE Press.

Blunden, B. (2009), "The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System", Jones and Bartlett Publishers, Inc, 1th edition.

Choi, Y., Kim, I., Oh, J. and Ryou, J. (2008). "PE File Header Analysis-Based Packed PE File Detection Technique (PHAD)", Proceedings of the International Symposium on Computer Science and its Applications, p.28-31, October 13-15, 2008.

Dinaburg, A., Royal, P., Sharif, M., and Lee, W. (2008). "Ether: Malware analysis via hardware virtualization extensions". In In Proceedings of The 15th ACM Conference on Computer and Communications Security (CCS 2008), Alexandria, VA, October 2008.

Father, H. (2004) "Hooking Windows API-Technics of Hooking API Functions on Windows", CodeBreakers J., vol.1, no.2, http://www.codebreakers-journal.com/downloads/cbj/2004/CBJ_1_2_2004_HolyFather_Hooking_Windows_API.pdf.

Franklin, J., Paxson, V., Perrig, A. and Savage, S. (2007). "An Inquiry Into the Nature and Causes of the Wealth of Internet Miscreants". In Conference on Computer and Communications Security (CCS), 2007.

Garfinkel, T. and Rosenblum, M. (2003) “A virtual machine introspection based architecture for intrusion detection”, In Proc. Network and Distributed Systems Security Symposium, p 191-206.

Hoglund, G. and Butler, J. (2005), “Rootkits: Subverting the Windows Kernel”, Addison-Wesley Professional, 1th edition.

Holz, T., Engelberth, M. and Freiling, F. (2008). “Learning More About the Underground Economy: A Case-Study of Keyloggers and Dropzones”. Reihe Informatik TR-2008-006, University of Mannheim, 2008.

JoeBox. (2010). http://www.joebox.org/, 2010.

Kang, M. G., Poosankam, P., and Yin, H., (2007). “Renovo: A hidden code extractor for packed exe-cutables”. In Proceedings of the 2007 ACM Workshop on Recurring Malcode (WORM 2007).

Kong, J. (2007), “Designing BSD Rootkits”, No Starch Press, 1th edition.

Leder, F. and Werner, T. (2009). “Know your enemy: Containing conficker”. http://www.honeynet.org/papers/. The Honeynet Project & Research Alliance.

Martignoni, L., Christodorescu, M., and Jha, S. (2007). “Omniunpack: Fast, generic, and safe unpack-ing of malware”. In Proceedings of the Annual Computer Security Applications Conference (ACSAC), 2007.

Microsoft PECOFF (2008), “Microsoft Portable Executable and Common Object File Format Specification”, http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/pecoff_v8.docx, March 2008.

Moser, A., Kruegel, C., and Kirda, E., (2007). “Limits of Static Analysis for Malware Detection”, In ACSAC, pages 421--430. IEEE Computer Society.

SecureList (2007), Net-Worm.Win32.Allaple.a, Kaspersky Labs, http://www.securelist.com/en/descriptions/old145521, August 2007.

SoftPanorama (2009), Network Worm Allaple.B, http://www.softpanorama.org/Malware/Malware_defense_history/Malware_gallery/Network_worms/allaple_rahack.shtml, December 2009.

Song , D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P. and Saxena, P. (2008). “BitBlaze: A New Approach to Computer Security via Binary Analysis”, Proceedings of the 4th International Conference on Information Systems Security, December 16-20, 2008, Hyderabad, India.

Willems, C., Holz, T. and Freiling, F. (2007). "Toward Automated Dynamic Malware Analysis Using CWSandbox," IEEE Security and Privacy, vol. 5, no. 2, pp. 32-39, Mar./Apr. 2007.

Yegneswaran, V., Saidi, H., Porras, P. (2008). “Eureka: A framework for enabling static analysis on malware”. Technical Report SRI-CSL-08-01 Computer Science Laboratory and College of Computing, Georgia Institute of Technology, April 2008.