UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

MALDETECT: UMA METODOLOGIA AUTOMATIZÁVEL DE DETECÇÃO DE *MALWARES* DESCONHECIDOS

LEANDRO SILVA DOS SANTOS

ORIENTADOR: Dr. DINO MACEDO AMARAL

DISSERTAÇÃO DE MESTRADO EM
ENGENHARIA ELÉTRICA

BRASÍLIA/DF: JUNHO/2016.

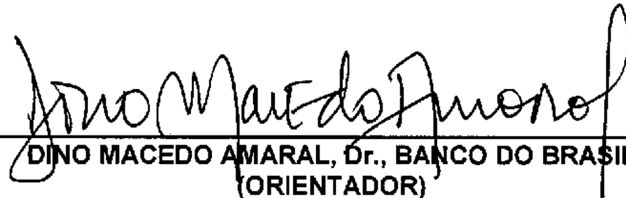
UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

MALDETECT: UMA METODOLOGIA AUTOMATIZÁVEL DE DETECÇÃO DE MALWARES DESCONHECIDOS

LEANDRO SILVA DOS SANTOS

DISSERTAÇÃO DE MESTRADO SUBMETIDA AO DEPARTAMENTO DE ENGENHARIA ELÉTRICA DA FACULDADE DE TECNOLOGIA DA UNIVERSIDADE DE BRASÍLIA, COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE.

APROVADA POR:



DINO MACEDO AMARAL, Dr., BANCO DO BRASIL
(ORIENTADOR)

ROBSON DE OLIVEIRA ALBUQUERQUE, Dr., ENE/UnB
(EXAMINADOR INTERNO)

LAERTE PEOTTA DE MELO Dr., BANCO DO BRASIL
(EXAMINADOR EXTERNO)

Brasília, 30 de junho de 2016.

FICHA CATALOGRÁFICA

SANTOS, LEANDRO SILVA DOS

Maldetect: Uma metodologia automatizável de detecção de *malwares* desconhecidos. [Distrito Federal] 2016.

xv, 95 p., 297 mm (ENE/FT/UnB, Mestre, Engenharia Elétrica, 2016).

Dissertação de Mestrado - Universidade de Brasília.

Faculdade de Tecnologia. Departamento de Engenharia Elétrica.

1. Maldetect — 2. Análise de memória

3. Detecção de *malwares* desconhecidos — 4. Volatility

I. ENE/FT/UnB — II. Título (série)

REFERÊNCIA BIBLIOGRÁFICA

Santos, L. S. (2016). Maldetect: Uma metodologia automatizável de detecção de *malwares* desconhecidos. Dissertação de Mestrado em Engenharia Elétrica, Publicação PPGENE.DM - 628/2016, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 95p.

CESSÃO DE DIREITOS

NOME DO AUTOR: Leandro Silva dos Santos.

TÍTULO DA DISSERTAÇÃO DE MESTRADO: Maldetect: Uma metodologia automatizável de detecção de *malwares* desconhecidos. GRAU / ANO: Mestre / 2016

É concedida à Universidade de Brasília permissão para reproduzir cópias desta dissertação de Mestrado e para emprestar ou vender tais cópias somente para propósitos acadêmicos e científicos. Do mesmo modo, a Universidade de Brasília tem permissão para divulgar este documento em biblioteca virtual, em formato que permita o acesso via redes de comunicação e a reprodução de cópias, desde que protegida a integridade do conteúdo dessas cópias e proibido o acesso a partes isoladas desse conteúdo. O autor reserva outros direitos de publicação e nenhuma parte deste documento pode ser reproduzida sem a autorização por escrito do autor.

Leandro Silva dos Santos
Rua 36 Norte, Lote 7, apto 1104, Águas Claras
71.919-180 Brasília - DF - Brasil.

DEDICATÓRIA

Dedico este trabalho ao autor da vida,
à minha amada esposa, à minha família
e aos meus amigos que me incentivaram,
e me ajudaram a vencer as barreiras.

AGRADECIMENTOS

Agradeço, primeiramente, à Deus por mais uma etapa concluída em minha vida, e à minha amada esposa que teve paciência e sempre me incentivou a vencer cada barreira.

Agradeço ao Felipe e ao Robson que me ajudaram a abrir as portas necessárias para iniciar esta caminhada.

Agradeço aos colegas de trabalho da PRDF que também foram importantes para que fosse possível a conclusão das aulas.

Agradeço ao meu orientador que dedicou seu tempo e conhecimento para me conduzir nessa caminhada para que eu pudesse alcançar o sucesso.

Agradeço ao Departamento de Engenharia Elétrica e seus funcionários que proporcionaram um ambiente propício aos estudos e ofereceram todo o suporte necessário para eu completasse essa etapa importante.

Leandro Silva dos Santos

RESUMO

MALDETECT: UMA METODOLOGIA AUTOMATIZÁVEL DE DETECÇÃO DE *MALWARES* DESCONHECIDOS

Autor: Leandro Silva dos Santos

Orientador: Dino Macedo Amaral

Programa de Pós-graduação em Engenharia Elétrica

Brasília, Junho de 2016

O cenário de ataques cibernéticos, acompanhando a modernização das ferramentas de detecção e remoção, tem se tornando cada vez mais complexo de ser detectado e mitigado. Com isso as ferramentas tradicionais de detecção e remoção de ameaças estão cada vez menos eficiente, principalmente por aquele seguimento que utiliza uma abordagem de detecção baseada em assinatura. Este trabalho propõe uma metodologia automatizável de detecção de *malwares* desconhecidos, ou seja, aqueles que não foram detectados pelas ferramentas tradicionais. A metodologia apresentada neste trabalho, denominada aqui por *Maldetect*, coleta e correlaciona características comportamentais típicas de códigos maliciosos, que estão presente no *dump* memória volátil, com o objetivo de identificar os artefatos que mais realizam atividades típicas de *malware*. Além disso, foi construída uma ferramenta usando as linguagens de programação PHP e Python, denominada *Maldetect Tool*, a qual automatiza a metodologia proposta. Esta ferramenta analisou *dumps* da memória volátil infectados com códigos maliciosos e gerou um relatório contendo os artefatos que mais realizaram atividades típicas de *malwares*.

ABSTRACT

MALDETECT:

Author: Leandro Silva dos Santos

Supervisor: Dr. Dino Macedo Amaral

Programa de Pós-graduação em Engenharia Elétrica

Brasília, June of 2016

The scenario of cyber attacks, following the modernization of detection and removal tools is becoming increasingly complex to be detected and mitigated. Thus the traditional tools of detection and removal of threats are becoming less efficient, especially by those that use a signature-based detection approach. This paper proposes an automatable method of detecting unknown malware, ie those that were not detected by traditional tools. The methodology presented in this work, called here by Maldetect, collects and correlates typical behavioral characteristics of malicious code, which are present in the volatile memory dump , in order to identify artifacts that most perform typical activities of malware. Moreover, it was built a tool using the languages of PHP and Python, called Maldetect Tool, which automates the proposed methodology. This tool analyzed the volatile memory dumps infected with malicious code and generated a report containing the artifacts held more typical activities of malware.

SUMÁRIO

1 INTRODUÇÃO 1

1.1 JUSTIFICATIVA E MOTIVAÇÃO 2

1.2 METODOLOGIA 3

1.3 OBJETIVOS 3

1.4 ORGANIZAÇÃO DA DISSERTAÇÃO 4

2 REVISÃO BIBLIOGRÁFICA 5

2.1 TRABALHOS RELACIONADOS 5

2.2 METODOLOGIAS DE ANÁLISE DE MEMÓRIA VOLÁTIL 6

2.2.1 METODOLOGIA DO SANS - *SYSTEM ADMINISTRATION, NETWORKING AND SECURITY* 7

2.2.1.1 Identificar processos estranhos 7

2.2.1.2 Analisar DLLs e *handles* de processos 8

2.2.1.3 Avaliar os artefatos de rede 9

2.2.1.4 Procura por evidências de injeção de código 9

2.2.1.5 Verificação de assinaturas de *hooking* 9

2.2.1.6 *Dump* de processos, DLLs e *drivers* suspeitos 10

2.2.2 BUSCA POR *MALWARES* NOS PROCESSOS EM MEMÓRIA 11

2.2.2.1 Recuperar linhas de comandos e caminho dos processos 11

2.2.2.2 Analisar *heaps* 11

2.2.2.3 Inspecionar variáveis de ambiente 12

2.2.2.4 Detectar *backdoors* com *handles* padrões 12

2.2.2.5 Enumerar DLLs 12

2.2.2.6 Extrair arquivos *portable executable* (PE) da memória 13

2.2.2.7 Detectar injeção de código 14

2.2.3 METODOLOGIA DE ANÁLISE DE MEMÓRIA 14
2.3 FERRAMENTAS DE ANÁLISE DE MEMÓRIA VOLÁTIL 15
2.3.1 Volatitlity Framework 15
2.3.1.1 Usando o *Volatility* e Comandos Básicos 16
2.3.1.2 Criando um *plugin* 17
2.3.2 Redline 17
2.3.2.1 Recursos usados na investigação 18
2.3.2.2 Passos de investigação 19
2.3.3 Comparação 20

3 METODOLOGIA MALDETECT 22
3.1 Pré-análise 22
3.2 Análise de processos 22
3.3 Análise de bibliotecas dinâmicas e *handles* 24
3.4 Análise de Artefatos de Rede 25
3.5 Busca por injeção de código 26
3.6 Busca por *hooks* 26
3.7 *Dump* de processos, bibliotecas dinâmicas e *drivers* suspeitos 27
3.8 Processamento, Correlação e Relatório 27
3.9 Criação da base de conhecimento 28

4 TÉCNICAS DE AUTOMATIZAÇÃO DA METODOLOGIA MALDETECT PARA WINDOWS 29
4.1 Pré-análise 29
4.2 Análise de processos 32
4.3 Análise de bibliotecas dinâmicas e *handles* 39
4.4 Análise de Artefatos de Rede 41
4.5 Busca por injeção de código 43

4.6 Busca por *hooks* 44

4.7 Dump de processos, bibliotecas dinâmicas e *drivers* suspeitos 47

4.8 Processamento, Correlação e Relatório 47

4.9 Criação da base de conhecimento 48

4.10 Módulos e outras considerações da implementação 48

5 RESULTADOS E DISCUSSÕES **51**

5.1 Ambiente de Laboratório 51

5.2 Procedimentos 52

5.3 Resultados 53

5.3.1 Análise de Processos 53

5.3.2 Análise de bibliotecas dinâmicas e *handles* 55

5.3.3 Análise de Artefatos de Rede 56

5.3.4 Busca por injeção de código 57

5.3.5 Busca por *hooks* 58

5.3.6 *Dump* de processos, bibliotecas e *drivers* suspeitos 59

5.3.7 Processamento, Correlação e Relatório 60

5.3.8 Criação da base de conhecimento 60

5.4 Resumo das análises 61

6 CONCLUSÕES E TRABALHOS FUTUROS **65**

REFERÊNCIAS BIBLIOGRÁFICAS **66**

APÊNDICES **70**

A Código do *plugin*: procinfo **71**

B Relatório do *malware* NF-e 18454310845.exe gerado pela ferramenta **76**

LISTA DE TABELAS

2.1 Comparação entre a Redline e o *Volatility* 20

4.1 Estrutura física da Tabela config do banco de dados MYSQL que armazena as informações coletadas na pré-análise 32

4.2 Verificação dos processos do Sistema Operacional Windows 7 (RUSSINOVICH; SOLOMON; IONESCU, 2012) (SANS, 2014) (OLSEN, 2014). 32

4.3 Estrutura física da tabela *process* do banco de dados MySQL que armazena as informações dos processos em execução no *dump* de memória em análise. 38

4.4 *Blacklist* de *mutexes* utilizada pela *Maldetect Tool* 40

4.5 Estrutura física da tabela DLL do banco de dados MySQL que armazena as informações das DLLs carregadas pelos processos em execução no *dump* de memória em análise. 41

4.6 Estrutura física da tabela nertwork do banco de dados MySQL que armazena as informações das atividades de rede encontradas no *dump* de memória em análise. 42

4.7 Tabela de porta consideradas normais em uma imagem de Windows 7 livre de *malwares* 43

4.8 Estrutura física da tabela known_db do banco de dados MySQL que armazena os XMLs dos IOCs dos artefatos maliciosos identificados durante a análise do *dump* de memória volátil. 48

4.9 Pontuação de anomalias detectadas pela *Maldetect Tool* 49

5.1 Resultado da submissão dos códigos maliciosos ao VirusTotal. 53

5.2 Atividades típicas de códigos maliciosos encontradas na fase de análise de processos e seus respectivos artefatos. O ranking é o somatório da pontuação das anomalias detectadas conforme a tabela 4.9 54

5.3 Atividades típicas de códigos maliciosos encontradas na fase de análise de bibliotecas dinâmicas e *handles* de processos. O ranking é o somatório da pontuação das anomalias detectadas conforme a tabela 4.9 55

5.4 Atividades típicas de códigos maliciosos encontradas na fase de análise de artefatos de rede. O ranking é o somatório da pontuação das anomalias detectadas conforme a tabela 4.9 56

5.5 Atividades típicas de códigos maliciosos encontradas na fase de busca por injeção de código. O ranking é o somatório da pontuação das anomalias detectadas conforme a tabela 4.9 57

5.6 Atividades típicas de códigos maliciosos encontradas na fase de busca por *hooks*. O ranking é o somatório da pontuação das anomalias detectadas conforme a tabela 4.9 58

5.7 Atividades típicas de códigos maliciosos encontradas na fase de *dump* de processos, bibliotecas e *drivers* suspeitos. O ranking é o somatório da pontuação das anomalias detectadas conforme a tabela 4.9 59

5.8 Atividades típicas de códigos maliciosos encontrados e seus respectivos artefatos. 61

LISTA DE FIGURAS

1.1 Gráfico de novos *malwares* detectados pelo instituto AV-TEST nos últimos 10 anos (AV-TEST, 2015). 3

2.1 Resultado da aquisição de memória com técnicas de antiforense ativada(STÜTTGEN; COHEN, 2013). 6

2.2 Metodologia de análise de memória do SANS, adaptado de (LEE, 2013). 7

2.3 Estrutura básica do cabeçalho de arquivos PE 13

2.4 *Volatility* em linha de comando. 16

2.5 *Volatility* em linha de comando. 18

3.1 Metodologia *Maldetect* 23

4.1 Tela inicial da versão beta da *Maldetect Tool*. 30

4.2 Arquitetura da *Maldetect Tool* para o caso de análise de memória volátil do Sistema Operacional Windows 7. O *plugin procinfo* não faz parte da lista de *plugins* do *Volatility* e foi criado especialmente para a automatização da metodologia *Maldetect*. 31

4.3 Exemplo de *hook* de IAT no processo exemplo.exe onde o endereço da função CreateFileW é substituído pelo endereço da função maliciosa. . 45

4.4 Exemplo de *hook* de EAT na DLL exemplo.dll onde o endereço da função exportada ReadFile é substituído pelo endereço da função maliciosa. . . 45

4.5 Exemplo de *inline hook* na DLL exemplo.dll onde o início da função exportada ReadFile é substituído pela instrução JMP do *assembly* que redireciona a execuçao para a função maliciosa. 46

4.6 Módulos de controle de cada fase da metodologia *Maldetect* e suas principais funções. 49

5.1 Topologia da rede do laboratório de análise automática de *dump* de memória. 52

LISTA DE SÍMBOLOS, NOMENCLATURA E ABREVIAÇÕES

AV: Antivírus.

APT: *Advanced Persistent Threats.*

TI: Tecnologia da Informação.

DFRWS: *Digital Forensic Research Workgroup.*

DLL: *Dynamic Link Library.*

IP: *Internet Protocol.*

URL: *Uniform Resource Locator.*

SANS: *System Administration Network and Security.*

SID: *Security Identifier.*

PID: *Process Identifier.*

PPID: *Parent Process Identifier.*

TCP: *Transmission Control Protocol* .

SSDT: *System Service Descriptor Table.*

IDT: *Interrupt Descriptor Table.*

IRP: *I/O Request Packet.*

PEB: *Process Enviroment Block.*

PE: *Portable Executable.*

CPL: *Control Panel Applets.*

IOC: Indicativo de Comprometimento.

SO: Sistema Operacional.

FTP: *File Transfer Protocol.*

XML: *eXtensible Markup Language.*

DNS: *Domain Name System.*

PDB: *Page Directory Bases.*

IAT: *Import Address Table.*

EAT: *Export Address Table.*

1 INTRODUÇÃO

O cenário de ataques cibernéticos, acompanhando a modernização das ferramentas de detecção e remoção, tem se tornando cada vez mais complexo de ser detectado e mitigado. A indústria de antivírus (AV) tem se demonstrado ineficiente contra ameaças avançadas, principalmente por aquele seguimento que utiliza detecção baseada em assinaturas. Este tipo de detecção pode ser burlada pelos *malwares* que utilizam técnicas de polimorfismos e metamorfismo (BAILEY et al., 2007). Apesar disso, o AV ainda possui seu espaço no arsenal de segurança da informação.

Inicialmente, os *malwares* tinham objetivos simples, como apagar arquivos ou provocar erros, ou ainda executar atividades indesejadas em um computador, as quais eram percebidas pelos usuários. Porém, com o avanço dos *malwares*, alguns tipos já são capazes de capturar, interceptar e até sequestrar dados relevantes das vítimas. Este último tipo de *malware* (chamados de *ransomware*(AJJAN, 2013)) criptografa os arquivos da vítima e pedem pagamento em dinheiro (geralmente utilizam a moeda digital *bitcoin*[1]) em troca da decifragem destes arquivos (BLUNDEN, 2011).

Existe ainda o conceito de *Advanced Persistent Threats* - APT (Ameaça Persistente Avançada), a qual geralmente possui alvos específicos e utiliza técnicas avançadas, como a exploração de uma ou mais vulnerabilidades *0-day* e o uso de certificados falsificados, para comprometer as estações de seus alvos(YANG; TIAN; DUAN, 2014). Assim, na maior parte dos casos, não são produzidos por indivíduos isolados, mas sim por instituições, crime organizado ou governos que mediante objetivos específicos ajudam a financiar tais atividades(BLUNDEN, 2011). Geralmente, este tipo de ameaça é usado em atividades de espionagem ou sabotagem (LI, 2011). São exemplo de ameaças avançadas persistentes:

* Stuxnet: seus alvos eram as usinas nucleares do Irã e tinha o objetivo de impedir ou dificultar que eles produzissem armas nucleares. Este código malicioso pode ter sido desenvolvido pelo governo dos Estados Unidos e de Israel (SANGER, 2012).

[1]https://bitcoin.org/pt_BR/

* Operação Aurora: foi uma séria de ataques que tinha como alvo mais de 30 grandes empresas, tais como: Google e Adobe. Estes ataques exploravam uma vulnerabilidade *0-day* do Internet Explorer (MACAFEE, 2010).

* Uroburos: este *rootkit* é composto por dois arquivos: um *driver* e um mini sistema de arquivos encriptado. Tinha como objetivo roubar informações e seu principal alvo era os Estados Unidos(DATA, 2014).

A forense de memória volátil é umas das principais técnicas utilizadas para analisar essas ameaças avançadas, pois é eficiente na identificação de características comportamentais típicas de *rootkits* e outros tipos de *malware*(STÜTTGEN; COHEN, 2013). Além disso, a análise de memória permite reconstruir o estado original do sistema no momento da coleta, identificar quais arquivos estão sendo acessados, as conexões de redes que foram abertas, dentre outros dados relevantes para a detecção de ações realizadas pelos códigos maliciosos(LIGH et al., 2010).

Dessa forma, este trabalho propõe uma metodologia automatizável de análise de memória volátil, denominada *Maldetect*. Esta é capaz de coletar caraterísticas comportamentais e correlacionar as informações de forma a identificar quais artefatos são os potenciais *malwares*. E com base nesta metodologia, foi construída uma ferramenta que automatiza a análise do *dump* de memória volátil do Sistema Operacional Window 7. Ao final é apresentado os resultados das análises realizadas por esta ferramenta.

1.1 JUSTIFICATIVA E MOTIVAÇÃO

O instituto AV-TEST independente de segurança de TI registra mais de 390 mil novos *malwares* por dia (AV-TEST, 2015). A figura 1.1 mostra o gráfico da quantidade de novos *malwares* que foram detectados por este instituto nos últimos dez anos. Os dados apresentados foram coletados até o dia 03 de junho de 2016, por isso o número de artefatos de 2016 é menor que o de 2015. Observando este aumento de criação de novos códigos malicioso, que são considerados *0-day*, fica claro que métodos de detecção baseado em assinaturas não são suficientes para proteger os atuais sistemas de informação. Dessa forma, este projeto apresenta uma metodologia de detecção de ameaças cibernéticas baseada na identificação de anomalias comportamentais, obtidas a partir da análise de memória volátil do Sistema Operacional.

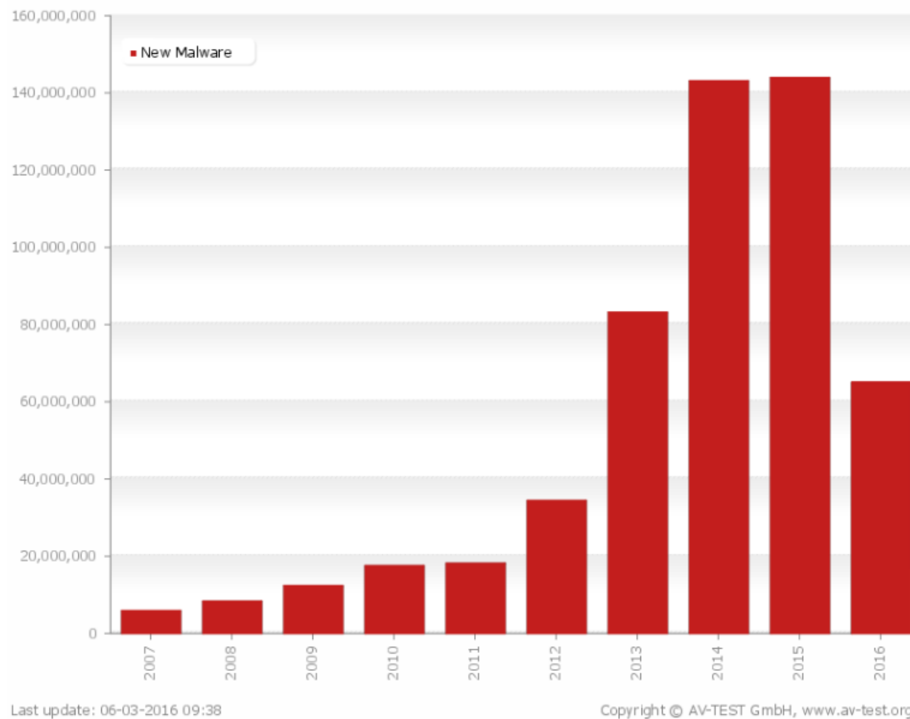


Figura 1.1: Gráfico de novos *malwares* detectados pelo instituto AV-TEST nos últimos 10 anos (AV-TEST, 2015).

1.2 METODOLOGIA

A metodologia do presente trabalho consistiu, em primeiro lugar, em fazer uma revisão bibliográfica referente ao tema proposto. Levantou-se as metodologias de análise de memória volátil existentes e trabalhos relacionados. As bibliografias levantadas serviram de base para a elaboração a metodologia proposta neste trabalho. Além disso, foi realizado um estudo de casos para construir o protótipo de uma ferramenta que automatizou tal metodologia.

1.3 OBJETIVOS

Este trabalho tem como objetivo elaborar uma metodologia automatizável de análise de memória volátil que seja capaz de detectar os artefatos que mais realizam atividades típicas de *malware*. Este objetivo geral será divido nos seguintes objetivos específicos:

* Estudar as metodologias existentes e os trabalhos relacionados;

* Propor uma nova metodologia baseada nas existentes, de maneira que suas fases possam ser realizadas de forma automatizada;

* Construir uma ferramenta que automatize a metodologia proposta;

* Utilizar a ferramenta construída para realizar a análise automatizada de *dumps* de memória volátil infectados com código malicioso.

1.4 ORGANIZAÇÃO DA DISSERTAÇÃO

Este trabalho está organizado da seguinte forma: o capítulo 2 apresenta as ferramentas e metodologias de análise de memória para detecção de artefatos maliciosos. O capítulo 3 detalha as fases da metodologia *Maldetect* para detecção de *malwares* desconhecidos a partir da análise do *dump* de memória volátil. O capítulo 4 descreve as técnicas utilizadas na construção de uma ferramenta que implementa a metodologia *Maldetect* para o Sistema Operacional Windows 7. O capítulo 5 apresenta os resultados obtidos a partir da execução da *Maldetect Tool* para detecção de artefatos maliciosos em *dumps* de memória volátil de estações infectadas e por fim, no último capítulo, são feitas as considerações finais e sugestões de trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA

2.1 TRABALHOS RELACIONADOS

Análise de memória volátil foi um dos principais temas do desafio de 2005 do Digital Forensic Research Workgroup (DFRWS), o que motivou um esforço de pesquisa e desenvolvimento de ferramentas nesta área(DFRWS, 2005). Este desafio deu início aos estudos de análise de memória usando técnicas forenses. Em (VÖMEL; FREILING, 2011), Vömel publicou um *survey* que apresenta várias técnicas de aquisição de memória baseada em software e hardware. Mostrando, também, várias técnicas de análise de processos, de recuperação de chave criptográfica, de análise de registro de sistemas, de redes, de arquivos, do estado do sistema e de uma aplicação específica baseada nas estruturas de memória do Sistema Operacional. Essas técnicas são amplamente utilizadas na análise manual de memória volátil.

Em (HU et al., 2013), *Liang Hu et al.* mostrou a importância de automatizar o processo de análise de memória volátil. O artigo propõe automatizar a análise de memória baseada em dois fluxos (*DLL flow* e *Process flow*). Em cada fluxo são coletados diversos dados que serão processados e correlacionados gerando um relatório. Porém, apenas os dados do fluxo de DLL são processados automaticamente.

Em (ESHAN, 2014), outra solução de automatização da análise de memória foi apresentada por Fahad - *Associate Director Security Research and Analytics UBS AG.* Ela é composta de três fases: primeiro é a aquisição da memória para um *drive* seguro que fique oculto para o usuário. Segundo, é a execução do Volatility[2] para extração das informações relevantes contidas no *dump* da memória. As duas primeiras fases serão executas a cada 30 minutos. Por fim, essas informações são enviadas para um servidor central que fará a análise. Esta fase executará um algoritmo de comparação do *dump* atual com as informações contidas na base de dados. Com essa abordagem, é possível identificar a criação de novas conexões de redes, novos serviços, alteração e criação de chaves de registros, entre outros dados. O problema é o aumento do processamento do *host* e o volume de dados sendo transferidos pela rede.

[2]Framework de análise de memória volátil. http://www.volatilityfoundation.org/

Acquisition tool	Version	Format	KDBG	MmGetPhysical memory-ranges()	MmMap-MemoryDump-Mdl()
Memoryze	2.0	raw	PASS	FAIL	PASS
FTK Imager	3.1.2	raw	PASS	FAIL	PASS
Win64dd	1.4.0	raw	PASS/FAIL	FAIL	FAIL
Win64dd	1.4.0	dmp	FAIL	FAIL	FAIL
DumpIt	1.4.0	raw	PASS	FAIL	FAIL
WinPmem	1.3.1	raw	FAIL	FAIL	PASS
WinPmem	1.3.1	dmp	FAIL	FAIL	PASS
WindowsMemoryReader	1.0	raw	PASS	FAIL	PASS
WindowsMemoryReader	1.0	dmp	PASS	FAIL	PASS

Figura 2.1: Resultado da aquisição de memória com técnicas de antiforense ativada(STÜTTGEN; COHEN, 2013).

Como a análise de memória tem sido amplamente aplicada na identificação de código malicioso, existem alguns cuidados que devem ser levados em consideração na fase de coleta da memória volátil, pois o processo de aquisição, geralmente, requer a execução de código na máquina infectada. Este processo pode ser interferido pelo *malware* em execução. Em (STÜTTGEN; COHEN, 2013), Johannes analisou várias técnicas antiforense que interferiam na aquisição da memória e testou as principais ferramentas com o objetivo de identificar quais delas seriam resistentes a estas técnicas. O resultado encontrado pode ser observado na figura 2.1 e nenhuma ferramenta foi resistente a todas as principais técnicas antiforense.

2.2 METODOLOGIAS DE ANÁLISE DE MEMÓRIA VOLÁTIL

No campo da computação forense, a análise de memória volátil pode trazer resultados mais eficazes na detecção de códigos malicioso do que a análise de artefatos de disco, já que a análise de memória identifica as ações que estão sendo executadas pelo Sistema Operacional e pelos aplicativos em execução no momento da coleta do *dump* da memória volátil.

Além disso, a análise de memória pode prover várias informações sobre o estado do sistema em tempo de execução, por exemplo: quais processos estão em execução, conexões de rede abertas e comandos executados recentemente. Os dados que ficam criptografados no disco, geralmente não estão criptografados ao serem carregados na memória volátil. Também é possível encontrar na memória chaves criptográficas, arquivos confidenciais e histórico dos *browsers* no modo de navegação anônima (LIGH et al., 2014).

Dessa forma, a seguir são descritas três metodologias de análise de memória volátil para detecção de artefatos maliciosos, as quais constituirão a base da metodologia proposta neste trabalho.

2.2.1 METODOLOGIA DO SANS - *SYSTEM ADMINISTRATION, NETWORKING AND SECURITY*

A metodologia do SANS de análise de memória volátil está focada na busca de artefatos maliciosos residentes em memória, ou seja, em execução no Sistema Operacional. Sua descrição não está concentrada em um único documento, mas é descrita em alguns casos de uso e *posters* públicos. Em (LEE, 2013) a metodologia é apresentada como sendo o nono passo da busca por *malwares* desconhecidos, como mostra a Figura 2.2.

Esta metodologia é composta de seis passos, alguns desses passos já são executados em uma análise padrão de memória, e outros são específicos para encontrar artefatos maliciosos. A análise de memória volátil nos fornece resultados capazes de identificar técnicas usadas por *rootkits*, os quais procuram dificultar sua detecção através da alteração do comportamento normal do Sistema Operacional. Assim, este tipo de código malicioso pode ocultar sua localização no sistema de arquivos e sua execução em memória.

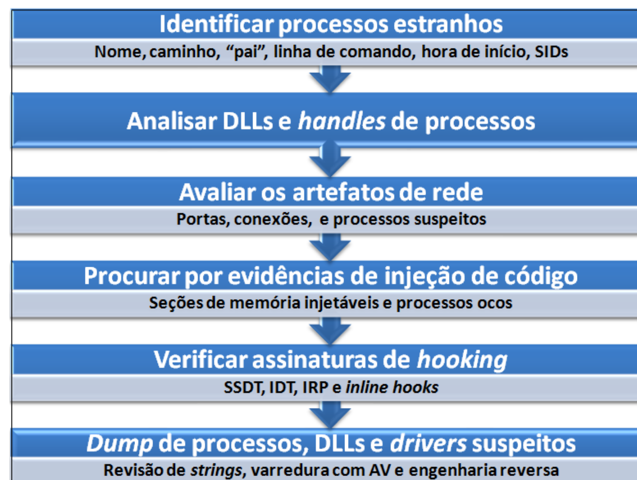


Figura 2.2: Metodologia de análise de memória do SANS, adaptado de (LEE, 2013).

2.2.1.1 Identificar processos estranhos

Na fase de análise de processos, devemos coletar algumas informações, tais como: nome do processo, caminho no sistema de arquivos do Sistema Operacional, processo "pai", linha de comando, hora de inicialização e SIDs (*security identifier*). Esses dados serão usados para: identificar processos legítimos; verificar a escrita correta do nome; identificar caminhos suspeitos dos processos; verificar o "pai" do processo; e identificar parâmetros de linha de comando que iniciou o processo. Um processo é denominado

processo “pai” (PPID- *parent pid*) quando ele cria outro processo (este é denominado processo “filho”) (WOODHULL; TANENBAUM, 2000).

2.2.1.2 Analisar DLLs e *handles* de processos

Para realizar a análise de DLLs e processos é preciso identificar as diferenças entre programa e processo. Um programa é uma sequência estática de instruções; já um processo contém um conjunto de recursos usados por uma instância de um programa.

Em (RUSSINOVICH; SOLOMON; IONESCU, 2012) são apresentados componentes de um processo do Sistema Operacional Windows:

* Um espaço de endereçamento virtual;
* Um programa executável;
* Um lista de *handles*[3] para vários recursos do sistema (por exemplo: portas de comunicação, semáforos e arquivos abertos);
* Uma lista de *Threads*;
* Um contexto de segurança; e
* Lista de DLLs (*dynamic-link libraries*) associadas.

Nesta fase, são analisados os *handles* associados aos processos de maneira a identificar alguma atividade maliciosa. Muitos *malwares* usam um *handle* do tipo *mutex*[4] (acrônimo para o termo em inglês, *mutal exclusion*, ou exclusão mútua) para identificar se o *malware* já foi instalado na máquina da vítima, e caso o *mutex* desse *malware* esteja presente na máquina o código malicioso não executa nenhuma tarefa. Além disso, deve-se verificar os caminhos em disco das DLLs, pois apesar de terem nomes legítimos podem estar armazenados em diretórios diferentes do padrão do Sistema Operacional.

[3]Referência abstrata para um recurso (RUSSINOVICH; SOLOMON; IONESCU, 2012).

[4]Mecanismo de sincronização usado para serializar o acesso à um recurso (RUSSINOVICH; SOLOMON; IONESCU, 2012).

2.2.1.3 Avaliar os artefatos de rede

Durante a análise dos artefatos de rede deve-se identificar as portas TCP suspeitas e os processos associados a elas, bem como os indicativos da presença de *backdoors* e a reputação dos IP's que a máquina está conectada. Dessa forma, podemos identificar atividades típicas de códigos malicioso com relação as conexões de rede(TILBURY, 2012).

2.2.1.4 Procura por evidências de injeção de código

Em (TILBURY, 2012), são apresentadas duas técnicas de injeção de código:

* *Injeção de DLLs*: muito utilizada pelos *malwares*, eles utilizam algumas chamadas de sistema, tais como: *VirtualAllocEx(), CreateRemoteThread e SetWindowsHookEx()* para carregar uma DLL no contexto de um processo já em execução;
* *Processos ocos*: o *malware* cria uma nova instância de um processo legítimo e substitui a área de código do processo legítimo pelo seu código malicioso e obtêm as DLLs, os *handles* e outros recursos do processo original.

A detecção de injeção de código pode ser feita varrendo a memória procurando por setores marcados com permissão de escrita e execução e que não tenham um mapeamento para um arquivo. Também pode ser feita uma comparação entre o código de memória e código do arquivo em disco para verificar o nível de similaridade(TILBURY, 2012).

2.2.1.5 Verificação de assinaturas de *hooking*

Hook pode ser definido como uma técnica utilizada por *rootkits* para modificar o comportamento normal de um módulo ou função de um processo. Este ataque pode ser a nível de *kernel* ou a nível de usuário dependendo do nível de permissão de execução do processo alvo do ataque.

Basicamente, existem quatro técnicas utilizadas pelos *rootkits* que devem ser verificadas nesta fase da metodologia. Deve-se verificar a *System Service Descriptor Table* (SSDT) - tabela que contém um *array* de ponteiros para as funções de tratamento de cada *system call*. As entradas da SSDT podem ser alteradas pelos *rootkits* e, assim, o código malicioso pode modificar a saída e/ou a entrada das chamadas de sistema e esconder processos, arquivos e/ou chaves de registros (RUSSINOVICH; SOLOMON; IONESCU, 2012).

Além disso, *drivers* maliciosos podem utilizar a *Interrupt Descriptor Table* - estrutura de dados que armazena os endereços das funções de tratamento de interrupção e exceções de processos - para realizar um *hook* em todas rotinas de tratamento ou em apenas um ponto, modificando o comportamento normal da rotina de tratamento de interrupção (LIGH et al., 2010).

Outra técnica utilizada é o *hook driver* (*Hooking the I/O Request Packet* - IRP), na qual os *rootkits* alteram a IRP - estrutura de dados que contém códigos para identificar as operações desejadas e *buffers* de dados que serão lidos ou escritos pelo *driver*. Geralmente, o módulo “tcpip.sys” é atacado com esta técnica(LIGH et al., 2010).

Por último, pode ser usada a *inline hook*, também conhecida como *Dynamic code patching*, que sobrescreve os primeiros *bytes* de um função com a instrução de *jump* (instrução JMP do *assembly* (HYDE, 2010)). Dessa forma, o fluxo de execução será redirecionado para a função do código malicioso, e ao final de sua execução este poderá retornar para a função original (OROSZLANY, 2008).

2.2.1.6 *Dump* de processos, DLLs e *drivers* suspeitos

Nesta fase, espera-se obter uma lista dos possíveis artefatos malicioso que precisam de uma análise mais profunda. Para esses possíveis *malwares* deve-se realizar um *dump* do processo correspondente da memória e realizar uma revisão de strings, varredura com antivírus, engenharia reversa e outras técnicas que possibilitem a detecção de atividades maliciosas.

2.2.2 BUSCA POR *MALWARES* NOS PROCESSOS EM MEMÓRIA

Em (LIGH et al., 2014), Michael Hale et al. descreve sete objetivos da análise de memória para se encontrar um *malware*, que são:

2.2.2.1 Recuperar linhas de comandos e caminho dos processos

O *Process Environment Blobk* (PEB), que é membro da estrutura de memória _EPROCESS[5], contém o caminho completo do processo, a linha de comando que iniciou o processo, ponteiros para a *heap* do processo, entre outras informações. Essas informações ajudam a localizar o arquivo executável no disco e descobrir informações sobre como o processo foi instanciado na memória da estação infectada.

2.2.2.2 Analisar *heaps*

A *Heap* de um processo é uma região de memória onde serão armazenadas variáveis alocadas dinamicamente. Quando não se sabe, antes da execução do processo, o tamanho da área de memória necessária para armazenar uma determinada estrutura de dados, este processo aloca dinamicamente uma região da área de *heap* para armazenar este dado. Na linguagem C, esta ação de alocação é realizada pela função *malloc* e a liberação de memória é realizada pela função *free*, ambas manipulam a *heap* do processo (COIMBRA, 2011).

Dessa forma, os dados que as aplicações manipulam (dados recebidos via rede, ou textos digitados em um processador de texto) possuem uma boa chance de estarem armazenados na *heap* do processo. Assim, quando se está em busca de uma informação específica manipulada pelo processo em análise, obtêm-se maior efetividade analisando a *heap* primeiro, antes de verificar as regiões de memória que contém DLLs, arquivos mapeados e a pilha.

[5] Estrutura de dados que armazena várias informações de um processo carregado em memória no Sistema Operacional Windows

2.2.2.3 Inspecionar variáveis de ambiente

Existem famílias de *malwares* que marcam sua presença com a criação de variáveis de ambiente. Outros *malwares* manipulam os valores das variáveis de ambientes para gerar comportamentos maliciosos em outros processos. Algumas variáveis que são tipicamente manipuladas por códigos maliciosos, são:

* PATH: armazena o caminho dos executáveis;
* PATHEXT: extensões atribuídas aos programas executáveis;
* Caminho dos diretórios temporários;
* Caminho dos diretórios de documentos, histórico de internet e dados de aplicações dos usuários;
* ComSpec: localização do *cmd.exe*;

2.2.2.4 Detectar *backdoors* com *handles* padrões

Através da análise dos *handles* dos processos, identifique se a entrada e saída de um processo estão sendo direcionados a um *socket* de rede remoto para um atacante. Uma técnica muito comum, usada pelos *backdoors*, é criação de um *socket* de rede associado a um processo "cmd.exe" de tal forma que toda saída do processo seja transmitido pela rede e todo dado recebido via *socket* de rede seja transformado em entrada para o processo.

2.2.2.5 Enumerar DLLs

Uma *Dynamic Link Libraries* (DLL) é conjunto de códigos e funções que podem ser exportadas e utilizadas por mais de um processo ao mesmo tempo. Dessa forma, essas bibliotecas promovem o uso eficiente da memória e a reutilização de códigos (MICROSOFT, 2014).

As *Dynamic Link Libraries* possuem códigos e recursos que podem ser compartilhados entre processos, por isso é comum entre os *malwares* a técnica de injetar DLLs em

processos legítimos. Esta técnica é capaz de criar uma *thread* no processo alvo cujo código pertence a DLL injetada. Assim, o código malicioso não cria um novo processo dificultando sua detecção. Durante a análise, deve-se verificar se existe alguma DLL não vinculada, se o caminho das DLLs no sistema de arquivos são adequados e o contexto em que as mesmas estão carregadas.

2.2.2.6 Extrair arquivos *portable executable* (PE) da memória

Portable Executable é o formato padrão dos arquivos binários executáveis do Sistema Operacional Windows. Este é o formato dos arquivos com as seguintes extensões: .exe (executáveis), .dll (*Dynamic Link Libraries*), .cpl (*Control Panel Applets*), .com (arquivode comandos), entre outros. A estrutura básica do cabeça dos arquivos PE é apresentado na figura 2.3

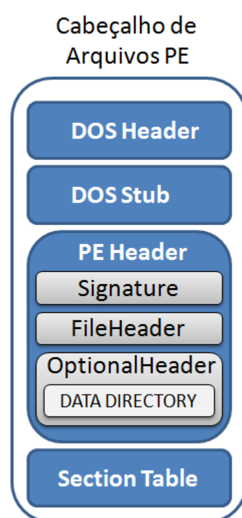


Figura 2.3: Estrutura básica do cabeçalho de arquivos PE

Estes arquivos PE, ou seja, arquivos executáveis, podem ser extraídos do *dump* de memória volátil em análise para uma verificação mais profunda deste artefato. Porém, um processo ao ser carregado na memória sofre algumas alterações que devem ser consideras durante a análise do artefato extraído. Por exemplo, o *hash* md5 do *dump* do processo extraído da memória pode não ser o mesmo *hash* do arquivo no disco. Mas, é possível usar um *fuzzy hash*(ZHANG; YIN; HAO, 2005) para determinar o grau de similaridade do arquivo extraído da memória com o arquivo original em disco.

2.2.2.7 Detectar injeção de código

A injeção de código é um tipo de ataque em que o código malicioso é capaz de adicionar um conjunto de instruções no processo alvo e forçar sua execução no contexto do processo vítima do ataque. A seguir são apresentados quatro técnicas de injeção de código:

* Injeção remota de DLLs: o processo malicioso força o processo alvo a carregar uma DLL específica;
* Injeção remota de código: o processo malicioso escreve código na área de memória do processo alvo e força sua execução;
* Injeção reflexiva de DLL: o processo malicioso escreve o código da DLL no espaço de memória do processo alvo; e
* Injeção em processo oco: o processo malicioso inicia uma nova instância de um processo legítimo em modo suspenso e então é feita uma sobrescrita da área de código do processo legítimo pelo código malicioso e sua execução é iniciada.

2.2.3 METODOLOGIA DE ANÁLISE DE MEMÓRIA

Em (MALIN; CASEY; AQUILINA, 2008), são descritos os objetivos da análise de memória, especificamente no contexto de análise de código malicioso:

- Coletar os metadados disponíveis, tais como: detalhes de processos, conexões de rede, e outras informações associadas ao potencial *malware*;
- Para cada processo de interesse, se possível, recuperar o arquivo executável da memória para análise; e
- Para cada processo de interesse extrair mais dados da memória, por exemplo, usuários, senhas e chaves criptográficas.

2.3 FERRAMENTAS DE ANÁLISE DE MEMÓRIA VOLÁTIL

Este capítulo apresenta as principais características do *Volatility Framework* e da Redline, que são ferramentas de análise de memória volátil. Apesar de existirem outras ferramentas que realizam esta análise, estas não possuem licença de gratuita para uso.

2.3.1 Volatitlity Framework

O *Volatility Framework* é uma coleção de ferramentas, implementada em Python, capaz de extrair artefatos digitais de um *dump* da memória volátil (RAM). O *Volatility* é licenciado pela *GNU General Public License 2*, possui código aberto e é gratuito. Sua arquitetura permite a inclusão de novas funcionalidades através da criação de novos *plugins* (LIGH et al., 2014).

O *Volatility* é capaz de analisar o *dump* de memória das versões 32-bits e 64-bits dos sistemas operacionais Windows, Linux, Mac e 32-bits do Android. O *Volatility* suporta a inclusão de novos sistemas operacionais devido a sua arquitetura modular. Porém, o *Volatility* não possui como objetivo ser uma ferramenta de aquisição de memória. Além disso, esta ferramenta não possui interface gráfica e seu uso é através de linha de comando (LIGH et al., 2014), como mostra a Figura 2.4.

As funcionalidades do *Volatility* são realizadas pelos *plugins* instalados na ferramenta. Através da criação de novos *plugins* é possível adicionar novos recursos tornando-a mais adaptável aos novos desafios de detecção de códigos maliciosos. Em (FOUNDATION, 2015), são apresentados os *plugins* do *Volatility* 2.4, os quais são agrupados na seguintes categorias:

* *Image Identification*: identificação do sistema operacional e suas estruturas de dados;
* *Processes and DLLs*: lista os processos e DLLs carregadas na memória;
* *Process Memory*: recupera informações específicas de um ou mais processos;
* *Kernel Memory and Objects*: lista e verifica componentes do *kernel*;
* *Networking*: recupera atividades de rede do *dump* da memória;

```
root@kali: ~
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
root@kali:~# vol -h
Volatility Foundation Volatility Framework 2.4
Usage: Volatility - A memory forensics analysis platform.

Options:
  -h, --help            list all available options and their default values.
                        Default values may be set in the configuration file
                        (/etc/volatilityrc)
  --conf-file=/root/.volatilityrc
                        User based configuration file
  -d, --debug           Debug volatility
  --plugins=PLUGINS     Additional plugin directories to use (colon separated)
  --info                Print information about all registered objects
  --cache-directory=/root/.cache/volatility
                        Directory where cache files are stored
  --cache               Use caching
  --tz=TZ               Sets the timezone for displaying timestamps
  -f FILENAME, --filename=FILENAME
                        Filename to use when opening an image
  --profile=WinXPSP2x86
                        Name of the profile to load
  -l LOCATION, --location=LOCATION
                        A URN location from which to load an address space
  -w, --write           Enable write support
```

Figura 2.4: *Volatility* em linha de comando.

* *Registry*: recupera dados armazenados nos registros do sistema operacional;

* *Crash Dump, Hibernation e Conversion*: executa o *parser* e analisa informações de arquivos de hibernação e *crash dumps*, bem como realiza a conversão entre esse tipos de arquivos; e

* *Miscellaneous*: agrupa os *plugins* de tipos diversos.

2.3.1.1 Usando o *Volatility* e Comandos Básicos

O arquivo principal para execução do *Volatility* é o *vol.py*. Como argumento de linha de comando para o *vol.py* pode ser passado o arquivo de *dump* de memória, o *profile* e o *plugin* que deseja executar. Segue o uso genérico da linha de comando:

```
1 $ python vol.py -f <FILENAME> --profile=<PROFILE> <PLUGIN> [ARGS]
```

A seguir um exemplo de uso do *Volatility* para execução do *plugin psscan* no arquivo de *dump* “\home \maldetect \dump.dmp” de um máquina com Windows 7 64bits:

```
$ python vol.py -f /home/maldetect/dump.dmp --profile=Win7Sp1x64 psscan
```

2.3.1.2 Criando um *plugin*

Para desenvolver um novo *plugin*, deve-se criar um classe em *Python*, a qual deve ser uma extensão da classe *commands.Command* e sobrescrever alguns métodos. Especificamente, deve ser sobrescrito dois métodos: *calculate* (neste método deve ser inserido o código que implementa a funcionalidade do *plugin*) e *render_text* (usado para formatar o resultado do *plugin*). Veja a seguir o código do *ExamplePlugin* (LIGH et al., 2014):

```
import volatility.utils as utils
import volatility.commands as commands
import volatility.win32.tasks as tasks

class ExamplePlugin(commands.Command):
        """Este é um plugin de exemplo"""

        def calculate(self):
                """Este método contém as instruções que este plugin irá
                    realizar"""


        def render_text(self, outfd, data):
                """Este método formata o resultado do plugin para ser
                    impresso na console de comando''''''
```

2.3.2 Redline

Redline é um ferramenta gratuita, produzida pela empresa Mandiant, e é usada para para detectar assinaturas de códigos maliciosos em *hosts* através da análise de memória volátil. Ela possui suporte apenas para o sistema operacional Windows. A figura 2.5 apresenta a tela inicial da interface gráfico da Redline.

Esta ferramenta auxilia o analista a manter o foco no alvo da investigação oferecendo um *workflow* de investigação e dados auxiliares para otimizar o tempo de análise. Primeiro apresentaremos os recursos auxiliares que podem ser utilizados durante a

Figura 2.5: *Volatility* em linha de comando.

análise e em seguida os passos de investigação sugerido pela ferramenta (MANDIANT, 2012).

2.3.2.1 Recursos usados na investigação

A ferramenta oferece três principais recursos que suportam as atividades de investigação de códigos maliciosos em memória volátil, os quais são descritos abaixo (MANDIANT, 2012):

* *Pontuação de Índice de Risco de Malware*: o índice de risco de *malware* é uma pontuação que a ferramenta atribui para os processos que são mais prováveis de terem sido atacados por um código malicioso. Esta pontuação não determina que o processo é um *malware*, mas identifica os processos que merecem mais atenção durante a investigação, o que geralmente otimiza a análise;

* *Indicativos de comprometimento (IOCs)*: a Redline utiliza um padrão aberto, desenvolvido pela própria Mandiant, para descrever as anomalias encontradas durante a análise um artefato malicioso. Este padrão de descrição também facilita a troca de informações com outras ferramentas;

* *Lista branca de MD5*: a Mandiant tem extraído milhões de *hashes* das várias versões do sistema operacional Windows e criado uma lista de *hashes* de arquivos conhecidos como livres de infecção maliciosa. Assim, esta lista pode ser usada durante a investigação para filtrar apenas arquivos suspeitos e assim otimizar a análise.

2.3.2.2 Passos de investigação

A ferramenta Redline sugere seis passos de investigação a serem seguidos, são eles(MANDIANT, 2012):

* *Revisão de processos baseado em uma pontuação de índice de risco de Malware*: nesta etapa um *ranking* dos processos é criado baseado no índice de risco do processo e assim a análise pode ser priorizada. Cada processo apontado pela ferramenta deve ser investigado manualmente para se determinar se é um código malicioso;

* *Revisão de conexões/portas de rede*: a ferramenta identifica uma lista de atividades de redes para que possam ser analisadas de forma a detectar atividades incomuns ou maliciosas;

* *Revisão de seções/DLLs em memória*: nesta etapa, são detectadas as seções de memórias atípicas e verifica-se se existem DLLs não assinadas e/ou que estão sendo usadas por apenas um processo. Geralmente, as DLLs legítimas são assinadas e usadas por vários processos. Se estiver usando a lista branca de MD5 é possível detectar DLLs que não pertencem a lista de arquivos conhecidos como livres de *malwares*;

* *Revisão de Handles não confiáveis*: por padrão, a Redline só apresenta *handles* que não são usados por pelo menos quatro processos confiáveis ou que são usados por poucos processos;

* *Revisão de Hooks*: a ferramenta detecta a lista dos *hooks* que foram considerados como não confiáveis. Porém estes devem ser investigados mais profundamente de maneira manual, já que é uma análise complexa para ser automatizada. Podem ser apresentados as seguintes categorias de *hooks*: *Untrusted Hooks, IDT Hooks, SSDT Hooks* e *IRP Hooks* ;

* *Revisão de Drivers e Devices*: nesta etapa é recomendado que seja feita uma análise manual dos *drivers*. Caso esteja utilizando a lista branca de MD5, esta etapa poderá ser otimizada.

2.3.3 Comparação

As duas ferramentas oferecem recursos úteis durante a análise de memória volátil, porém possuem algumas diferenças que são apresentas na tabela 2.1.

Tabela 2.1: Comparação entre a Redline e o *Volatility*

Característica	Redline	*Volatility*
Open source	Não	Sim
Multiplataforma	Não	Sim
Possui Interface gráfica	Sim	Não
Possui *whitelist* de *hashes* conhecidos	Sim	Não
Possui *workflow* de investigação	Sim	Não
Execução via linha de comando	Não	Sim
Extensão de funcionalidades por meio de *plugins*	Não	Sim

Para este projeto optou-se por utilizar o *Volatility Framework*, pois os recursos desta ferramenta pode ser estendido através da criação de novos *plugins*, o que permite adicionar e customizar nova funcionalidades. Dessa forma, é possível recuperar atributos específicos dos artefatos contidos no *dump* de memória volátil para realizar as etapas da metodologia proposta.

Além de ser uma ferramenta multiplataforma desenvolvida em *Python*, sua execução através de linha de comando permite automatizar a execução de suas funcionalidades. O que facilita a captura dos resultados via *script*.

Também, é uma ferramenta robusta que suporta analisar o *dump* de memória volátil de vários sistemas operacionais. Assim como, é possível criar novos perfis para que a ferramenta seja capaz de interpretar novos tipos de sistemas operacionais.

Por outro lado, a ferramenta Redline possui um *workflow* de investigação semelhante as metodologias existentes apresentadas no capítulo anterior. Além disso, também possui importantes conceitos, como: indicativos de comprometimentos e pontuação de índice de riscos de *malware*. Este conceitos foram incorporados na metodologia proposta que será apresentado no próximo capítulo.

3 METODOLOGIA MALDETECT

Este capítulo descreve a metodologia *Maldetect* que se assemelha a metodologia do *SANS* descrita no capítulo 2. Esta metodologia apresenta conceitos que podem ser aplicados a qualquer S.O. A *Maldetect* coleta e correlaciona informações comportamentais dos artefatos (processos, bibliotecas dinâmicas, drivers e outros) residentes no *dump* de memória volátil e identifica quais características são típicas de códigos maliciosos. Para melhor entendimento, a figura 3.1 apresenta um resumo de cada fase da metodologia proposta e a seguir estas serão descritas em detalhes.

3.1 Pré-análise

A pré-análise é uma fase de preparação e otimização da metodologia. Possui como objetivo coletar dados relevantes, os quais são específicos do *dump* de memória a ser analisado, a fim de parametrizar a análise a ser realizada. A partir destes dados, pode ser criada uma linha base contendo as atividades consideradas não-nocivas ao funcionamento do sistema. Por exemplo, para analisar de um servidor de banco de dados deverá ser coletada qual a porta de conexão de rede que a aplicação está sendo executada, pois, senão estas conexões serão consideradas anômalas durante a análise do *dump* de memória volátil.

3.2 Análise de processos

Para se realizar as verificações necessárias desta fase da metodologia, algumas informações devem ser coletadas de cada processo em execução no *dump* de memória volátil em análise. Esses dados devem ser no mínimo os seguintes: nome, caminho completo, PID, PPID, linha de comando de inicialização, hora de inicialização, hora de término, processos filhos, prioridade de execução, dono do processo, sessão em que está sendo executado, número de *threads* em execução e número de *handles*.

Além disso, deve-se conhecer a lista dos processos do núcleo do Sistema Operacional a ser analisado, assim como diversas características consideradas não-nocivas destes

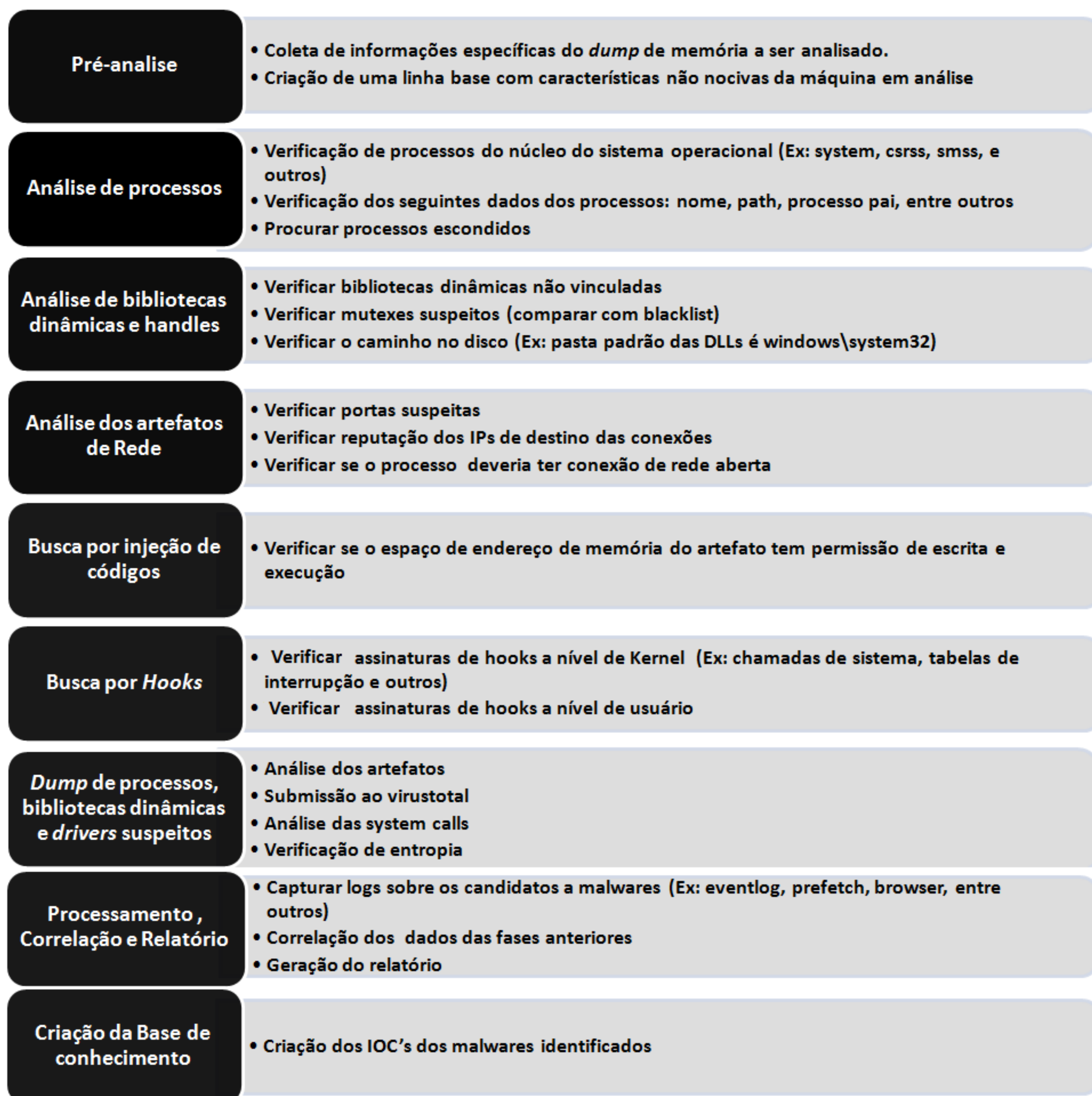


Figura 3.1: Metodologia *Maldetect*

processos. A partir desses dados, a metodologia *Maldetect* filtra a lista dos processos do núcleo do Sistema Operacional do *dump* da memória volátil a ser analisado e compara se, neste caso, as características destes artefatos apresentam alguma anomalia. Para cada anomalia detectada, deverá ser armazenada sua descrição e os artefatos envolvidos receberão uma pontuação de acordo com o tipo de comportamento anômalo identificado. Esta pontuação será processada na fase apropriada da metodologia.

Ainda nesta fase, devem ser usadas técnicas que percorrem a lista de processos em execução por caminhos diferentes, com o objetivo de encontrar quais processos utilizam técnicas de ocultação (VÖMEL; FREILING, 2011), ou seja, processos que apesar de estarem em execução não seriam listados no gerenciador de tarefas do Sistema Operacional.

Por fim, deve-se verificar se os binários estão sendo executados a partir de pastas temporárias e se existem processos com nomes similares aos processos do núcleo do Sistema Operacional, visto que muitos códigos maliciosos usam nomes com grafias erradas para tentar passar desapercebido como um processo não-nocivo do Sistema Operacional.

3.3 Análise de bibliotecas dinâmicas e *handles*

Durante a análise das bibliotecas dinâmicas, que estavam em memória no momento da captura da memória volátil, deve-se verificar se há alguma que não possui vínculo, ou seja, aquelas que não fazem parte da lista de *threads* de um processo. Além disso, será identificada a existência de bibliotecas sendo executadas diretamente no contexto de um processo hospedeiro que tem apenas o objetivo de fazer a carga da biblioteca em memória.

Também, devem ser analisados os contextos nos quais as bibliotecas foram carregadas, ou seja, verificar se o objetivo da biblioteca tem correspondência com o objetivo geral do processo. Essa verificação permite detectar processos que tipicamente não deveriam realizar conexões de rede, mas possuem *threads* de bibliotecas dinâmicas que são usadas para realizar atividades de rede.

Outra característica a ser verificada, é a localização no sistema de arquivos no qual as bibliotecas estão armazenadas, a fim de identificar quais estão em pastas temporárias

ou suspeitas. Ainda nesta fase, a metodologia *Maldetect* requer a verificação da escrita correta do nomes de bibliotecas previamente conhecidas como parte do núcleo do Sistema Operacional.

Por fim, os *handles* do tipo *mutexes* devem ser comparados com uma *blacklist* de nomes usados por *malwares* já conhecidos. Geralmente, este é um recurso muito utilizado pelos códigos maliciosos para identificar se a máquina já foi infectada. Também, deve-se identificar a existência de *pipes* (mecanismo que redireciona a saída de uma programa como entrada de outro) redirecionando entradas e saídas de um processo para um programa remoto, pois esta técnica é muito usada por *backdoors* (LIGH et al., 2014).

3.4 Análise de Artefatos de Rede

O acesso a rede é utilizado pelos *malwares* para diferentes fins, tais como: extravio de informações, *download* de novos componentes, comunicação com a central de comandos, infecção de novas vítimas, disparo de envio de e-mails em massa e ataques de negação de serviços. Dessa forma, deve-se listar as portas de rede que estão em modo *listening* do protocolo TCP e verificar quais delas são consideradas como suspeitas. Para realizar esta análise, a metodologia *Maldetect* prevê um conhecimento prévio de quais portas são não-nocivas para o Sistema Operacional que está sendo analisado, estas portas são armazenadas em uma *whitelist*. Assim, todas as portas listadas no *dump* de memória volátil que não estão nesta *whitelist* serão consideradas como suspeitas e os processo ligados a estas portas recebem uma pontuação que e será processada na fase correspondente.

Nesta fase, as informações coletadas na pré-análise ajudam na identificação das portas suspeitas e não-nocivas. Caso a máquina em análise seja um servidor FTP e na pré-análise tenha sido informado que a porta 21 do protocolo TCP em modo *listening* é uma porta não-nociva, então durante a análise o artefato que fez o *bind* na porta 21 não será considerado anômalo. Porém, caso estas informações não sejam conhecidas previamente deve-se considerar as portas que não pertencem ao funcionamento normal do Sistema Operacional como anomalias.

Além disso, deve-se realizar uma verificação dos IP's de destino de conexões estabelecidas com uma *blacklist* de IP's maliciosos. E por fim, deve-se verificar se existem

interfaces de rede em modo promíscuo, assim como, quais processos estão fazendo uso das conexões de rede e se os mesmos deveriam fazer uso deste recurso.

3.5 Busca por injeção de código

As técnicas usadas nesta etapa devem ser capazes de identificar as assinaturas de injeção de código, tal como a injeção de bibliotecas dinâmicas e sobrescrita de código *assembly* em memória. Os artefatos que possuem áreas de memória marcadas como READ/WRITE/EXECUTE são pontuados como suspeitos para serem correlacionados com outras anomalias, pois são os possíveis alvos de ataques de injeção de código. Além disso, deve-se verificar a existência de processos ocos, os quais foram descritos no capítulo 2.

3.6 Busca por *hooks*

Hook é uma técnica utilizados pelos códigos maliciosos para modificar o comportamento normal de uma função. Esta técnica pode ser utilizada em nível de *kernel* e em nível de usuário. Portanto, nesta fase, deve-se verificar a existência de ambos os tipos de *hooks*. No nível de *kernel* deve-se verificar, pelo menos, os seguintes:

- Alterações na tabela que armazena os ponteiros para as funções de tratamento das chamadas de sistema;
- Modificações no vetor que armazena os ponteiros para as funções de tratamento de interrupções do Sistema Operacional;
- Inserção ou substituições de instruções *assembly* no início do código da função alvo do ataque; e
- Modificações nas áreas de *buffers* dos *drivers*.

Já no nível de usuário deve-se verificar a tabela de importação e de exportação de funções do artefato, identificando se os endereços contidos nestas tabelas estão dentro da faixa de endereço da biblioteca que possui a função importada. Ainda nesta fase,

deve-se verificar a existência de execução de programas em modo *debug* e a existência de *threads* órfãs. Todas as anomalias identificadas recebem uma pontuação que será correlacionada e processada na fase apropriada.

3.7 *Dump* de processos, bibliotecas dinâmicas e *drivers* suspeitos

Essa etapa da metodologia tem o objetivo de aprofundar a análise dos possíveis códigos maliciosos identificados nas outras fases. Estes artefatos serão reconstruídos a partir do *dump* de memória que está sendo analisado. Os artefatos podem ser: processos, *drivers* ou bibliotecas dinâmicas. Após a extração do código binário, deve-se calcular o *hash* MD5 ou SHA256 destes arquivos e ser comparados com uma base de *hash* de *malwares* já conhecidos, como o *Virustotal*.

Além disso, outra verificações devem ser realizadas, tais como:

- Identificação de *strings* suspeitas (URL, endereços IPs, email, CPF e nomes de arquivos de sistema);
- Identificação de chamadas de sistemas comuns entre os *malwares*;
- Identificação de nomes suspeitos das seções do *assembly*; e
- Cálculo da entropia dos arquivos para identificação de técnicas de ofuscação de código (LYDA; HAMROCK, 2007).

3.8 Processamento, Correlação e Relatório

Os dados coletados e armazenados nas fases anteriores são correlacionados nesta fase e o relatório da análise é gerado. O relatório deverá apresentar todos os artefatos que receberam pontuação nas fases anteriores em ordem decrescente, de forma que os artefatos que estão no topo da lista possuem maior probabilidade de ser um código malicioso. Além disso, deverá apresentar cada anomalia identificada como típica de *malware*.

Para complementar o relatório, outras informações relativas a estes possíveis *malwares* podem ser coletadas a partir de outra fontes. Essas fontes podem ser histórico de navegadores, *log* do Sistema Operacional, lista de arquivos recentemente acessados, anomalias de *timeline*, histórico de comandos executados e data de compilação do arquivo executável.

3.9 Criação da base de conhecimento

Na última fase, deve-se utilizar padrões de descrição de indicativos de comprometimentos (IOC) para alimentar a base de conhecimento. Como exemplo, podem ser gerados IOC's baseado no *framework* OpenIOC (*framework open source* capaz de descrever as características comportamentais dos *malwares*(LOCK, 2013)). Este padrão descreve caraterísticas comportamentais que permite identificar um *malware*. Além disso, permite o compartilhamento de informações em uma linguagem comum.

No caso da *Maldetect*, algumas verificações não estão descritas nestes padrões abertos e, por isso, será usado um padrão próprio para descrição dos IOC's encontrados. Os IOC's detectados serão descrito em um arquivo XML que permite documentar as anomalias encontradas de um determinado artefato. Um exemplo deste arquivo será apresentado na seção 5.4.

Nesta fase, o arquivo XML gerado em cada análise poderá ser comparado com a base de conhecimento e assim verificar se as anomalias detectadas já existem na base. Dessa forma, será possível identificar artefatos diferentes que realizam os mesmos comportamentos anômalos e inferir se estes artefatos são da mesma familia e/ou são produzidos pelos mesmo grupo de pessoas.

4 TÉCNICAS DE AUTOMATIZAÇÃO DA METODOLOGIA MALDETECT PARA WINDOWS

Neste trabalho, atendendo aos requisitos da metodologia *Maldetect*, foi construída uma ferramenta que implementa cada fase da metodologia de forma a automatizar as tarefas de detecção de artefatos maliciosos residentes em *dumps* de memórias voláteis. A figura 4.1 mostra a tela inicial desta ferramenta, denominada de *Maldetect Tool*, a qual foi implementa nas linguagens *PHP* e *Python*. Para um melhor entendimento do funcionamento desta ferramenta, a figura 4.2 mostra a arquitetura e interação da *Maldetect Tool* com os recursos externos. Os principais recursos são: *Volatility*, *VirusTotal*, *IPVoid*[6] e a base de conhecimento.

Foi criado um repositório no github (https://github.com/maldetect/maldetect) para disponibilizar os relatórios contendo os resultados das análises realizadas com a *Maldetect Tool*. A seguir serão descritas as técnicas utilizadas na construção da *Maldetect Tool* para o estudo de caso da análise de memória volátil do Sistema Operacional Windows 7.

4.1 Pré-análise

Esta fase tem o objetivo de coletar informações para otimizar a execução da análise do *dump* de memória. Nesta fase, a *Maldetect Tool* coleta estas informações e configura a ferramenta armazenando os seguintes dados:

* Image Path: este campo deve ser preenchido com localização no sistema de arquivo onde está armazenado o *dump* de memória a ser analisado. O formato do *dump* deve ser compatível com os formatos aceitos pelo *Volatility*.

* Profile: é usado para armazenar o Sistema Operacional e sua arquitetura. Deve ser usado o padrão aceito pelo *Volatility*. Como a *Maldetect Tool* foi construída

[6]IPVoid é um serviço *online* e gratuito que faz análise de IP e DNS baseado em *blacklist*. Acessado em http://www.ipvoid.com/

Maldetect - Detecting Unknown Malware

Fases

Pré-analise e Configuração
Image Path: /root/maldetect/dumps/win7-
Profile: Win7SP1x64
System root: c:\windows
Base Directory Volatility: /root/maldetect/volatility/win7
Wget Directory: /root/maldetect/wget/win7-se
Network Ports (Ex. 80,443): 80
Salvar

Fase 1: Análise de Processos
Executar

Fase 2: Análise de DLLs e Handles
Executar

Fase 3: Análise dos Artefatos de Rede
Executar

Fase 4: Busca por Injeção de Códigos
Executar

Fase 5: Busca por Hooks
Executar

Fase 6: Dump de Processos, DLLs e Drivers Suspeiros
Executar

Fase 7: Processamento, Correlação e Relatório
Executar Gerar Relatório

Fase 8: Criação da Base de Conhecimento
Executar

Autor: Leandro Silva dos Santos

Figura 4.1: Tela inicial da versão beta da *Maldetect Tool*.

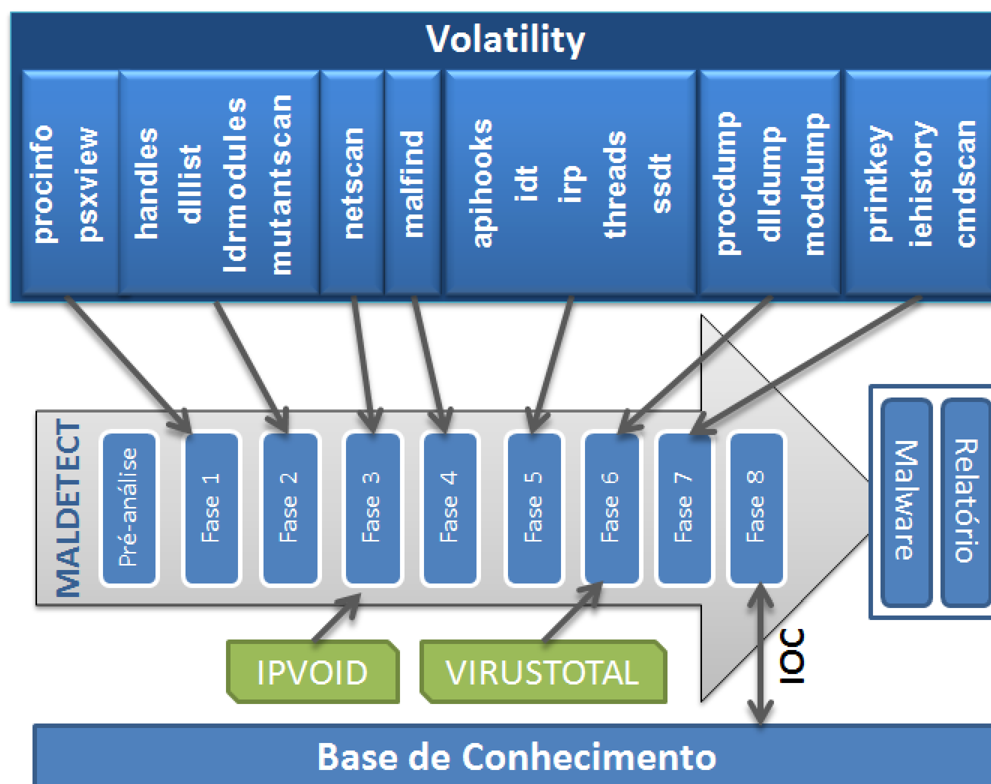


Figura 4.2: Arquitetura da *Maldetect Tool* para o caso de análise de memória volátil do Sistema Operacional Windows 7. O *plugin procinfo* não faz parte da lista de *plugins* do *Volatility* e foi criado especialmente para a automatização da metodologia *Maldetect*.

apenas para o Sistema Operacional Windows 7 serão aceitos apenas os seguintes: Win7SP0x86, Win7SP0x64, Win7SP1x86, Win7SP1x64.

* System root: é o caminho no sistema de arquivos onde o Sistema Operacional Windows 7 está instalado na máquina a ser analisada.

* Base Directory Volatility: diretório onde os arquivos temporários gerados pelo *Volatility* serão armazenados durante a execução da *Maldetect Tool*

* Wget Volatility: diretório onde os arquivos temporários gerados pelo Wget serão armazenados durante a execução da *Maldetect Tool*

* Network Ports: é usado para armazenar as portas de redes do protocolo TCP que serão consideradas como não-nocivas ao sistema. Caso seja necessário inserir mais de uma porta, basta separar por vígula, conforme exemplo: 80,443,8080.

Os dados coletados nesta fase são armazenados em uma tabela no banco de dados MySQL que possui os campos descritos na tabela 4.1.

Tabela 4.1: Estrutura física da Tabela config do banco de dados MYSQL que armazena as informações coletadas na pré-análise

Coluna	Tipo
id	int(11)
image	text
profile	varchar(100)
system_root	text
dir_base_volatility	text
wget_dir	text
network_port	varchar(200)

4.2 Análise de processos

Para executar as tarefas desta fase da metodologia *Maldetect* é necessário conhecer quais são os processos do núcleo do Sistema Operacional. No caso do Windows 7, os principais processos analisados pela *Maldetect Tool* são: *System*, *Smss*, *Csrss*, *Wininit*, *Services*, *Lsass*, *Svchost*, *Lsm*, *Winlogon*, *Explorer*, *Conhost*, *Rundll32*, *Taskhost*, *IExplore* e *CMD*.

Para cada um desses processos um conjunto de verificações são realizadas para determinar quais características não estão de acordo com um sistema livre de código malicioso. A tabela 4.2 apresenta o conjunto de verificações realizada pela *Maldetect Tool*.

Tabela 4.2: Verificação dos processos do Sistema Operacional Windows 7 (RUSSINOVICH; SOLOMON; IONESCU, 2012) (SANS, 2014) (OLSEN, 2014).

Processo	**Verificação**

System	Procura por processos que possuem nomes similares; Apenas um processo deve estar em execução; Verificar se o PID é 4; Verificar se o PPID é 0; Este processo deve possuir um filho e o nome do processo filho deve ser smss.exe.
smss.exe	Procura por processos que possuem nomes similares; Apenas um processo deve estar em execução; Verificar se o processo "pai" é o processo system e o PID do pai deve ser 4; O *base priority* do processo deve ser 11; O *username* do processo deve ser S-1-5-18 (Local System) O caminho do processo em disco deve ser $SYSTEMROOT\system32; Este processo não deve possuir filhos; O parâmetro *session running* deve ser 0.
csrss.exe	Apenas um processo pode estar em execução por *session running*; Procura por processos que possuem nomes similares; O caminho do processo em disco deve ser $SYSTEMROOT\system32; O *username* do processo deve ser S-1-5-18 (Local System); Os processos csrss.exe não devem possuir "pai"; O processo csrss.exe que executa na *session running* zero deve possuir *base priority* 13.
wininit.exe	Procura por processos que possuem nomes similares; Apenas um processo deve estar em execução; Não deve possuir processo "pai"; O *username* do processo deve ser S-1-5-18 (Local System); Este processo possui três filhos: lsass.exe, lsm.exe e services.exe; O *base priority* deve ser 13; O caminho do processo em disco deve ser $SYSTEMROOT\system32; O *session running* deve ser zero.
services.exe	Procura por processos que possuem nomes similares; Apenas um processo deve estar em execução; O processo "pai" deve ser o wininit.exe; O *username* do processo deve ser S-1-5-18 (Local System) O *session running* deve ser zero; O caminho do processo em disco deve ser $SYSTEMROOT\system32;

lsass.exe	Procura por processos que possuem nomes similares; Apenas um processo deve estar em execução; O processo "pai" deve ser o wininit.exe; O *username* do processo deve ser S-1-5-18 (Local System) O *session running* deve ser zero; O caminho do processo em disco deve ser $SYSTEMROOT\system32; O *base priority* deve ser 9.
svchost.exe	Procura por processos que possuem nomes similares; O processo "pai" deve ser o services.exe; O caminho do processo em disco deve ser $SYSTEMROOT\system32; O *username* deve ser um dos seguintes: S-1-5-18 (Local System), S-1-5-19 (NT Authority/Local Service) ou S-1-5-20 (NT Authority/Network Service); O *base priority* deve ser 8; A linha de comando deve possuir o parâmetro -k.
lsm.exe	Procura por processos que possuem nomes similares; Apenas um processo deve estar em execução; O processo "pai" deve ser o wininit.exe; Este processo não deve possuir filhos; O caminho do processo em disco deve ser $SYSTEMROOT\system32; O *base priority* deve ser 8; O *username* do processo deve ser S-1-5-18 (Local System); O *session running* deve ser zero.
winlogon.exe	Procura por processos que possuem nomes similares; Apenas um processo deve estar em execução; Este processo não deve possui "pai" em execução; O *username* do processo deve ser S-1-5-18 (Local System); Este processo pode não possuir filhos, mas se tiver será o userinit.exe; O *base priority* deve ser 13;O *session running* deve ser um; O caminho do processo em disco deve ser $SYSTEMROOT\system32;

explorer.exe	Procura por processos que possuem nomes similares; Apenas um processo deve estar em execução; Este processo pode não possuir “pai”, mas se tiver será o userinit.exe; O *username* será o usuário da sessão; O caminho do processo em disco deve ser $SYSTEMROOT\ O *base priority* deve ser 8.
conhost.exe	Procura por processos que possuem nomes similares; O processo “pai” deve ser o csrss.exe; O caminho do processo em disco deve ser $SYSTEMROOT\system32.
rundll32.exe	Procura por processos que possuem nomes similares; O caminho do processo em disco deve ser $SYSTEMROOT\system32.
iexplore.exe	Procura por processos que possuem nomes similares; O caminho do processo em disco deve ser \Programs Files\Internet Explorer ou \Programs Files (x86)\Internet Explorer
cmd.exe	Procura por processos que possuem nomes similares; Normalmente o “pai” é o explorer.exe; O caminho do processo em disco deve ser $SYSTEMROOT\system32;
outros processos	Verificar quais processos estão sendo executados a partir de pastas temporárias, *downloads*, users e appdata.

Segundo (FOUNDATION, 2015), o *Volatility* possui três *plugins* para extrair a lista de processos em execução na memória volátil no momento da captura, os quais são descritos a seguir:

* *pslist*: percorre a lista de processos e mostra os seguintes atributos dos processos: *offset*, *process name*, *process ID*, *parent process ID*, número de *threads*, número de *handles*, data e hora de início e fim do processo. Este *plugin* não é capaz de mostrar processos ocultos.

* *pstree*: exibe a lista de processos em formato de árvore e mostra os seguintes atributos: *process name*, *process ID*, *parent process ID*, número de *threads*, número de *handles*, data e hora de início e fim do processo.

* *psscan*: este *plugin* é capaz de mostrar processos inativos ou processos ocultos.

Ele mostra os seguintes atributos: *offset*, *process name*, *process ID*, *parent process ID*, *Page Directory Base* (PDB), data e hora de início e fim do processo.

Dessa forma, os *plugins* já existentes no *Volatility* não retornam todas as informações consideradas relevantes para as verificações que devem ser realizadas nesta fase. Então, com o intuito de implementar esta análise, foi necessária a construção de um novo *plugin* em *Python* para o *Volatility*, que foi denominado como *procinfo*. O *plugin* construído extrai os seguintes atributos da lista de processos: PID, PPID, PROCESS_NAME, BASE-PRIORITY, PATH, COMMAND_LINE, SESSIONID, CREATE_TIME, EXIT_TIME, HANDLES, THREADS e USERNAME.

A partir destes atributos, foi possível realizar todas as verificações que estão na tabela 4.2, as quais, de acordo com a metodologia são necessárias para se detectar comportamentos anômalos nos processos residentes em memória no momento da captura. O código completo deste *plugin* pode ser encontrado no Apêndice A, segue abaixo parte do código do *plugin* construído que mostra a coleta dos atributos retornado pelo *procinfo*:

```
1  class ProcInfo (commands.Command):
2
3          def __init__(self, config, *args):
4                  """Este função registra o parâmetro -o na linha de
                       comando para que o offset seja informado
5                          Uso do plugin sem offset: vol -f [dump] --profile
                              =[profile] procinfo
6                          Uso do plugin com offset: vol -f [dump] --profile
                              =[profile] procinfo --offset=[offset]
7                  """
8                  commands.Command.__init__(self, config, *args)
9                  self._config.add_option('offset', short_option='o',
                       default=None,
10                                 help='Physical Address', type='int')
11
12
13         def calculate (self):
14                 """Esta função obtém um ponteiro para lista de processos,
                        e no caso do offset ser informado obtém um ponteiro
                       para estrutura de memória do processo"""
15                 config= self._config
16                 addr_space = utils.load_as(self._config)
17                 if (config.offset != None):        """Caso o offset seja
                       infomado"""
```

```
18                  offset = taskmods.DllList.
                        virtual_process_from_physical_offset(
                        addr_space, config.offset).obj_offset
19                  yield obj.Object("_EPROCESS", offset = offset, vm
                        = addr_space)
20          else:   """Caso o offset não seja informado"""
21                  for proc in tasks.pslist(addr_space):
22                          yield proc
23
24      def render_text(self, outfd, data):
25              """Esta função imprimi o resultado"""
26              outfd.write("PID ;| PPID ;| PROCESS_NAME ;| BASEPRIORITY
                    ;| PATH ;| COMMAND_LINE ;| SESSIONID ;| CREATE_TIME ;|
                    EXIT_TIME ;| HANDLES ;| THREADS ;| USERNAME\n")
27              for proc in data:
28                      token = proc.get_token()
29                      sids = list(token.get_sids())
30                      sid_string = sids[0]
31                      if sid_string in well_known_sids:
32                          sid_name = " ({0})".format(well_known_sids[
                                sid_string])
33                      else:
34                          sid_name_re = find_sid_re(sid_string,
                                well_known_sid_re)
35                          if sid_name_re:
36                              sid_name = " ({0})".format(sid_name_re)
37                          else:
38                              sid_name = ""
39
40                      outfd.write("{0} ;| {1} ;| {2} ;| {3} ;| {4} ;|
                            {5} ;| {6} ;| {7} ;| {8} ;| {9} ;| {10} ;|
                            {11}\n".format(proc.UniqueProcessId, proc.
                            InheritedFromUniqueProcessId, proc.
                            ImageFileName,proc.Pcb.BasePriority,proc.Peb.
                            ProcessParameters.ImagePathName, proc.Peb.
                            ProcessParameters.CommandLine, proc.Peb.
                            SessionId,proc.CreateTime, proc.ExitTime, proc
                            .ObjectTable.HandleCount, proc.ActiveThreads,
                            sid_string + sid_name))
41
42
43  }
```

Para facilitar o acesso às informações coletadas pelo *plugin procinfo*, foi construído

um módulo em *PHP*, denominado de *process_analyser*, que realiza um *parser* destas informações e as carrega em uma tabela do banco de dados MySQL. A estrutura da tabela do banco de dados que armazenas essas informações obtidas do *dump* de memória, denominada *process*, é apresentada na tabela 4.3.

Tabela 4.3: Estrutura física da tabela *process* do banco de dados MySQL que armazena as informações dos processos em execução no *dump* de memória em análise.

Coluna	Tipo
pid	int(11)
ppid	int(11)
process_name	int(11)
base_priority	varchar(100)
username	varchar(200)
path	varchar(250)
create_sessions	varchar(30)
session_running	int(11)
time_create	timestamp
time_exited	timestamp
threads	int(11)
cmd_line	mediumtext
ranking	int(11)
anomaly_information	longtext
hidden	varchar(20)
offset	varchar(70)
log_information	text

Após a carga no banco de dados, os valores desses atributos são verificados se estão de acordo com a documentação da Microsoft conforme mostra a tabela 4.2. Para cada divergência encontrada, os processos envolvidos recebem uma pontuação de acordo com o tipo de anomalia. Os valores desta pontuação estão na tabela 4.9.

Ainda nesta fase, devem ser usadas técnicas para detectar processos ocultos (*hidden process*), para tal foi utilizado o *plugin psxview* do *Volatility*. Este *plugin* acessa a lista de processos da estrutura _EPROCESS de maneiras diferentes, e por isso, tem a capacidade de detectar um processo não vinculado. O *plugin procinfo* não é capaz de identificar um processo oculto, porém, através do *offset* do processo oculto obtido pelo *plugin psxview*, o *procinfo* é capaz de extrair todas as informações de um processo oculto.

4.3 Análise de bibliotecas dinâmicas e *handles*

Para executar as tarefas desta fase, foi construído um módulo para a *Maldetect Tool* que possui as seguintes funções: *checkMutex*, *checkHandleBackdoor*, *loadDLLs*, *checkNetwotkDLL*, *checkDLLPath* e *checkUnlikedDLL*. As funções realizam as seguintes verificações:

* *checkMutex*: para realizar a verificação de *mutex* foi criada uma *blacklist* para consulta. Essa lista pode ser incrementada a medida que várias análises forem realizadas e novos *mutexes* forem detectados como identificadores de códigos maliciosos. O *mutex* é muito usado pelos *malwares* para identificar se a máquina já foi contaminada. Em (SEGER, 2014), foi apresentado que o *mutex* "2gvwnqjz1" é muito usado pelos *malwares*, pois em todos os casos onde foi encontrado estava associado a um código malicioso. A *Maldetect Tool* utiliza o *plugin mutantscan* do *Volatility* para extrair a lista de *mutex* utilizado por cada processo. Caso algum dos *mutexes* pertencentes a um processo esteja na *blacklist* de *mutexes* maliciosos este processo recebe a pontuação correspondente a anomalia detectada, conforme tabela 4.9. A tabela 4.4 apresenta a *blacklist* de *mutexes* utilizada pela *Maldetect Tool*.

* checkHandleBackdoor: primeiro foi realizada uma busca na tabela de processos por todos os processos com nome de "cmd.exe" e em seguida executou-se o *plugin handles* do *Volatility* para cada processo "cmd.exe" em execução, a fim de encontrar características típicas de códigos maliciosos. Caso alguns desses processos possua um *handle* do tipo *File* com valor "*Device**Afd**Endpoint*", pois é um comportamento comum dos *Backdoors*(LIGH et al., 2014), este processo receberá a pontuação correspondente a esta anomalia detectada, conforme tabela 4.9.

Tabela 4.4: *Blacklist* de *mutexes* utilizada pela *Maldetect Tool*

Mutex	Família de código malicioso associado
2gvwnqjz1	diversas
__AVIRA__	ZEUS
svchost_test_started	TLD3
Flameddos	Bifhost
__b4ng__b4ng__38	Tigger
__SYSTEM__	ZEUS
Jo1ezds1	Bankpatch.C
Op1mutx9	Sality
Ap1mutx7	Sality
exeM_	Sality
Jhdheddfffffjk5trh	Allaple
1337bot	Spybot
Rootz	Sdbot
__Dassara__	jackal
)!VoqA.I4	Poison Ivy

* loadDLLs: através do *plugin dlllist* do *Volatility*, listou-se todas as DLLs carregadas estaticamente por cada processo e usando o *ldrmodules* do *Volatility* foi possível listar as DLLs carregadas dinamicamente pelos processos, ou seja, aquelas que foram carregadas pela função de sistema *LoadLibrary*. Armazenou-se estas informações numa tabela do banco de dados MySQL. A Tabela 4.5 apresenta a estrutura da tabela que armazena as informações de DLLs extraídas do *dump* de memória em análise.

* checkNetwotkDLL: as DLLs winsock32.dll, ws2_32.dll, wininet.dll e urlmon.dll tipicamente são utilizadas por processos que realizam conexões de rede no Sistema Operacional Windows 7. Então, a *Maldetect Tool* lista quais processos carregaram alguma dessas DLLs e permite que o usuário identifique quais processos tipicamente não fazem acesso a rede e mesmo assim estão utilizando alguma des-

Tabela 4.5: Estrutura física da tabela DLL do banco de dados MySQL que armazena as informações das DLLs carregadas pelos processos em execução no *dump* de memória em análise.

Coluna	Tipo
dll_id	int(11)
pid	int(11)
dll	text
ranking	int(11)
anomaly_description	text
base	varchar(120)

tas DLLs. Dessa forma, estes processos carregaram uma DLL fora de contexto e será atribuída uma pontuação correspondente a essa anomalia, conforme tabela 4.9.

* checkDLLPath: esta função realiza uma consulta no banco de dados para encontrar quais DLLs não estão armazenadas na pasta padrão do Sistema Operacional. No caso do Sistema Operacional Windows 7 a pasta padrão é "windows\system32", se for 64bits a pasta padrão é "windows\SysWOW64". Estas DLLs são listadas em uma tabela onde o usuário da *Maldetect Tool* pode marcá-las como DLL em caminho suspeito. Além disso, esta função atribui uma pontuação de anomalia para DLLs que estão armazenadas no sistema de arquivos em caminhos que contenham as palavras "appdata", "users", "downloads" e "temp", conforme tabela 4.9.

* checkUnlikedDLL: para verificar quais DLLs não estão vinculadas foi utilizado o *plugin ldrmodules* do *Volatility*, o qual percorre a lista de DLLs de formas diferentes.

4.4 Análise de Artefatos de Rede

As informações de conexões de rede foram extraídas do *dump* de memória em análise através o *plugin netscan* do *Volatility*. Estas informações foram armazenadas em uma tabela do banco de dados MySQL. A figura 4.6 apresenta a estrutura desta tabela.

Além disso, foi feito um estudo em um *dump* de memória do Sistema Operacional Windows 7 livre de infecção de códigos maliciosos para se determinar quais portas de rede podem ser consideradas não-nocivas. Dessa forma, foram identificadas as seguintes portas apresentadas na tabela 4.7.

Tabela 4.6: Estrutura física da tabela nertwork do banco de dados MySQL que armazena as informações das atividades de rede encontradas no *dump* de memória em análise.

Coluna	Tipo
id	int(11)
protocol	varchar(10)
local_ip	text
local_port	varchar(10)
remote_ip	text
remote_port	varchar(10)
state	varchar(30)
pid	int(11)
status	text

Após a carga das informações no banco de dados, as porta informadas pelo usuário na pré-análise e as atividades de rede consideradas não-nocivas são marcadas como normais. A *Maldetect Tool* analisa as outras atividades de rede em duas etapas: portas em modo *Listening* e conexões estabelecidas. As funções que realizam respectivamente estas etapas são: *checkLocalPort* e *checkRemoteIP*. A seguir serão descritas as verificações realizadas por estas etapas:

* checkLocalPort: nesta função, as portas do protocolo TCP que estão em estado *Listening*, e que não fazem parte da *whitelist* de portas do Windows 7 apresentada na tabela 4.7 e, também, não foram informadas na fase de pré-análise, serão consideradas como suspeitas e os processos que estão associados a elas recebem a pontuação correspondente com a anomalia detectada, conforme tabela 4.9.

* checkRemoteIP: ao consultar o banco de dados pode-se obter a lista de todas as atividades de rede que não estão em modo *Listening*. A partir desta lista,

Tabela 4.7: Tabela de porta consideradas normais em uma imagem de Windows 7 livre de *malwares*

Porta	Processo
135	svchost.exe
139	System
445	System
3702	svchost.exe
5355	svchost.exe
5357	System
49152	wininit.exe
49153	svchost.exe
49154	svchost.exe ou lsass.exe
49155	svchost.exe ou lsass.exe
49156	services.exe
54272	svchost.exe
57505	svchost.exe
57506	svchost.exe

cada IP remoto de uma conexão é consultado em base de reputação de IP's para determinar se a máquina está se comunicando com uma estação conhecida como maliciosa. A *blacklist* utilizada pela *Maldetect Tool* é a base do IPVoid. Caso algum dos IP's consultados retornem como suspeitos, o processo associado àquela conexão recebe a pontuação da anomalia encontrada, conforme tabela 4.9.

4.5 Busca por injeção de código

A *Maldetect Tool* utiliza o *plugin malfind* do *Volatility* para identificar as área de memórias são passíveis de serem vítimas de injeção de código. Este *plugin* mapeia as áreas de memória que estão marcadas como PAGE_EXECUTE_READWRITE, as quais possuem permissão de escrita e execução. Assim, os processos associados a

essas áreas são marcados como suspeitos e recebem a pontuação correspondente. Este *plugin* também detecta processos ocos, caso o atacante não tenha alterado as *flags* de permissão da área de memória substituída pelo código malicioso.

4.6 Busca por *hooks*

A especificação da metodologia *Maldetect* prevê a detecção de *hooks* a nível de usuário e a nível de *kernel*. No caso do Sistema Operacional Windows, o *hook* a nível de usuário ocorre através da alteração da *import address table* (IAT), da *export address table* (EAT) e o *inline hook*. O IAT e o EAT fazem parte do cabeçalho *portable executable* (PE). A IAT armazena o nome da função, o nome da DLL e o endereço da memória que armazena a função. Este endereço de memória pode ser alterado pelo atacante e substituído pelo endereço de memória do código malicioso, conforme mostra a figura 4.3.

De forma similar ocorre no *hook* do EAT, porém o endereço alterado é uma função exportada pelo módulo vítima do código malicioso, conforme mostra a figura 4.4. Por fim, no *inline hook* a primeira instrução do código *assembly* de um função exportada pelo módulo atacado pelo código malicioso é sobrescrita por uma instrução JMP do *assembly* para o endereço de memória da função do *malware*.

Para detectar estes três tipos de *hooks* a *Maldetect Tool* utiliza o *plugin apihooks*. Para o caso do *hook* de IAT o *plugin* verifica se o endereço de memória de cada entrada da tabela de importação pertence à faixa de endereço de memória do módulo correspondente da função importada. Para detectar alterações na EAT, o *plugin* verifica se o endereço de cada função exportada pertence à faixa de endereço de memória do módulo que exporta tais funções. Além disso, verifica se o início dessas funções possuem um *jump* (instrução JMP do *assembly*) para um endereço de memória que não pertence ao *range* de endereço de memória do módulo. Vale ressaltar que estes três tipos de *hook* podem ser classificados como *hook* a nível de *kernel*, caso o módulo atacado esteja sendo executado com permissão de *kernel*.

Os principais tipos de *hooks* a nível de *Kernel* do Sistema Operacional windows 7 são as alterações na *System Service Descriptor Table* (SSDT), *I/O Request Packets* IRP e *Interrupt Descriptor Table* (IDT). A SSDT é um tabela que armazena endereços de funções exportadas pelo *kernel* do Sistema Operacional, o *plugin ssdt* do *Volatility*

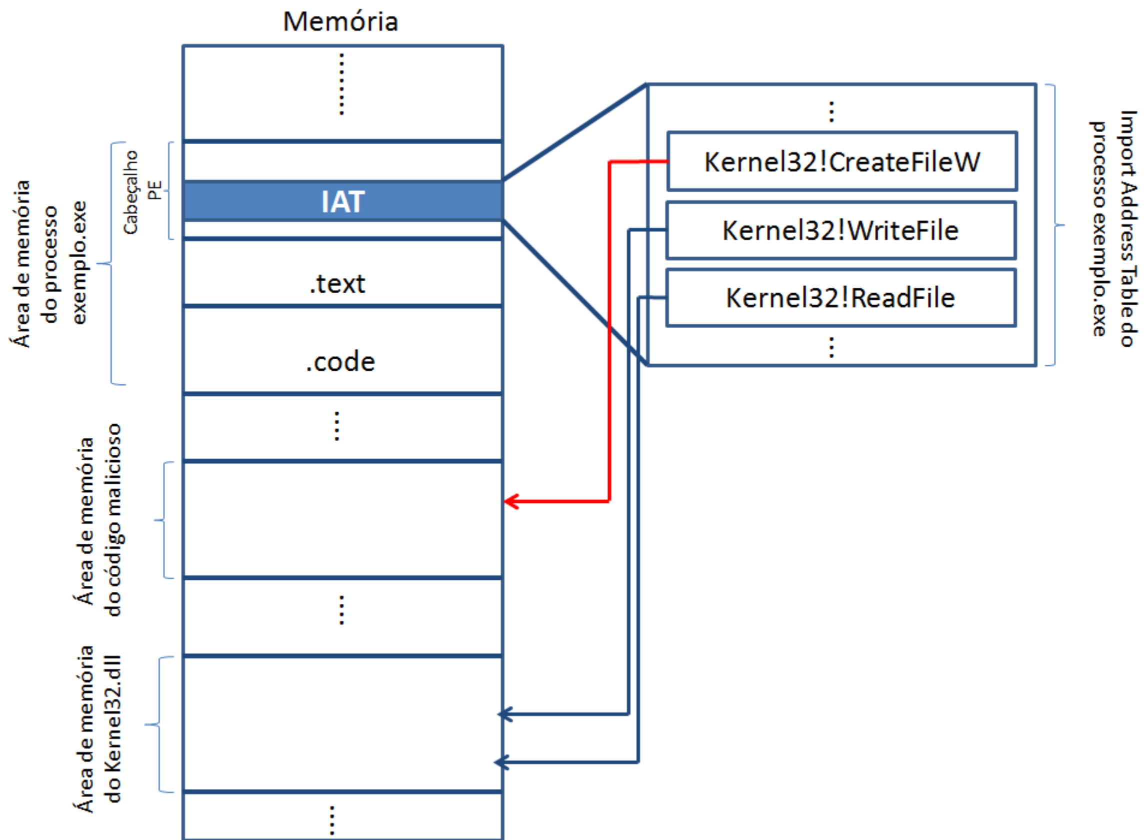


Figura 4.3: Exemplo de *hook* de IAT no processo exemplo.exe onde o endereço da função CreateFileW é substituído pelo endereço da função maliciosa.

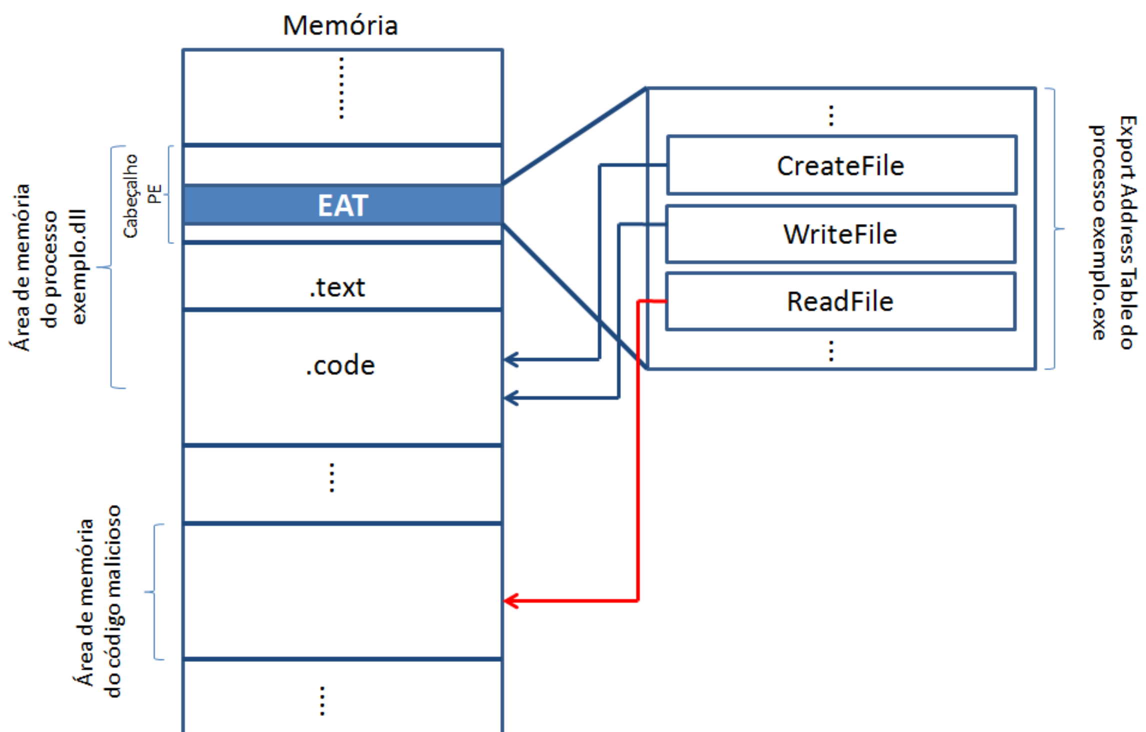


Figura 4.4: Exemplo de *hook* de EAT na DLL exemplo.dll onde o endereço da função exportada ReadFile é substituído pelo endereço da função maliciosa.

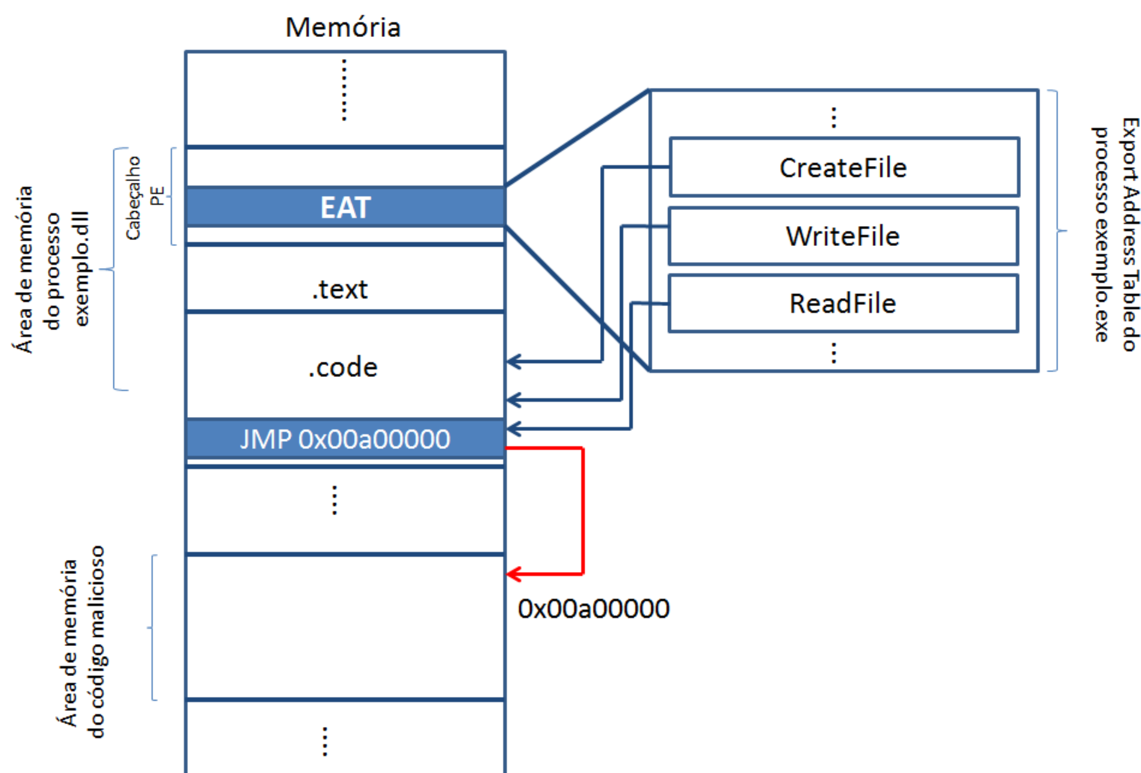


Figura 4.5: Exemplo de *inline hook* na DLL exemplo.dll onde o início da função exportada ReadFile é substituído pela instrução JMP do *assembly* que redireciona a execuçao para a função maliciosa.

verifica se os endereços armazenados nesta tabela estão dentro do range de memória correto. Os endereços contidos nesta tabela precisam estar dentro do *range* de memória do "ntoskrnl.exe" e "win32k.sys". Os *plugins irp* e *idt* do *Volatility* detectam os outros dois tipos de *hooks* verificando os endereços armazenados no vetor de tratamento de interrupção e *buffer* de dados dos *drivers*.

Por fim, nesta fase a *Maldetect Tool* utiliza o *plugin threads* do *Volatility* para verificar se existem *threads* órfãs em execução no *dump* de memória em análise. A *thread* orfã é um técnica usada pelo código malicioso para se ocultar no sistema, uma *thread* orfã é uma *thread* que se desvinculou do módulo original que criou a *thread*.

A automatização da análise de *hooks* com objetivo de diferenciar um *hook* malicioso de um não-nocivo é uma tarefa difícil, já que esta técnica também é usada por processos legítimos. Por isso, a *Maldetect Tool* identifica como um *hook* suspeito aqueles realizados por artefatos que já receberam pontuação maior ou igual a 5. Assim, essa fase só deve ser executada ao final das análises das fases anteriores.

4.7 Dump de processos, bibliotecas dinâmicas e *drivers* suspeitos

Nesta fase, são listados os processos em ordem decrescente do *ranking* de anomalias encontradas nas fases anteriores. Assim, o analista pode escolher os artefatos que serão extraídos da memória para uma análise mais aprofundado neste artefato. Para realizar a extração destes artefatos foi utilizado os seguintes *plugins* do *Volatility*: *procdump*, *ddldump* e *moddump*. Após a extração, é calculado o *hash* sha256 do artefato obtido e utilizando a API em python do *Virustotal* é realizada uma consulta ao banco de dados do *Virustotal* para determinar se existe alguma referência ao *hash* em questão. Para cada antivírus que detecta o artefato com um código malicioso, este artefato recebe um ponto no *ranking* de anomalias.

4.8 Processamento, Correlação e Relatório

Todas as informações obtidas nas fases anteriores são processadas e correlacionadas de tal forma que os possíveis códigos maliciosos recebam uma pontuação para cada anomalia encontrada, conforme 4.9. Nesta fase, os processos que carregaram uma DLL considerada como suspeita terão sua pontuação acrescida do valor da pontuação da DLL. Por exemplo, caso o ranking do processo seja 15 e possui vínculo com duas DLLs com as seguintes pontuações: 5 e 3. Ao final do processamento o processo terá pontuação final igual a 23.

Quanto maior a sua pontuação, mais anomalias o processo possui. A descrição de cada anomalia detectada nas fases anteriores foi armazenada na base de dados para fins de relatório. Além disso, para compor o relatório final a *Maldetect Tool* coleta as seguintes informações:

* valores da chave de registro Microsoft\Windows\CurrentVersion\Run (usada para armazenar programas que serão executados junto com a inicialização do Windows);
* histórico de acesso do Internet Explorer;
* histórico de comandos do cmd.exe.

Para capturar essas informações foram utilizados respectivamente os seguintes *plugins* do *Volatility*: *printkey*, *iehistory* e *cmdscan*. A *Maldetect Tool* gera um relatório em PDF contendo todos os artefatos que receberam uma pontuação de anomalias, assim como as anomalias de rede, o histórico de comandos do "cmd.exe" e acessos do Internet Explorer. O Apêndice B apresenta um relatório gerado pela ferramenta.

4.9 Criação da base de conhecimento

Na última fase da metodologia, o analista pode escolher quais processos ele gostaria de gerar o arquivo de IOC para compor a base de conhecimento. Este arquivo possui o formato XML onde são descritas as anomalias encontradas durante as fases anteriores da metodologia. A base de conhecimento é uma tabela no banco de dados MySQL que armazena o XML gerado nesta fase. Além disso, pode ser realizada uma busca pelo IOC armazenado na base para averiguar se o artefato identificado já foi identificado como código malicioso em análises anteriores. A tabela 4.8 mostra a estrutura da base de conhecimento.

Tabela 4.8: Estrutura física da tabela known_db do banco de dados MySQL que armazena os XMLs dos IOCs dos artefatos maliciosos identificados durante a análise do *dump* de memória volátil.

Coluna	Tipo
id_known_db	int(11)
ioc	text
date	date

4.10 Módulos e outras considerações da implementação

A *Maldetect Tool* possui um módulo de controle para cada fase da metodologia, conforme mostra a figura 4.6. A figura também mostra as principais funções desses módulos e cada função é responsável pela verificação de uma característica comportamental típica de códigos maliciosos.

Além disso, a medida que cada função de verificação detecta uma anomalia, o artefato

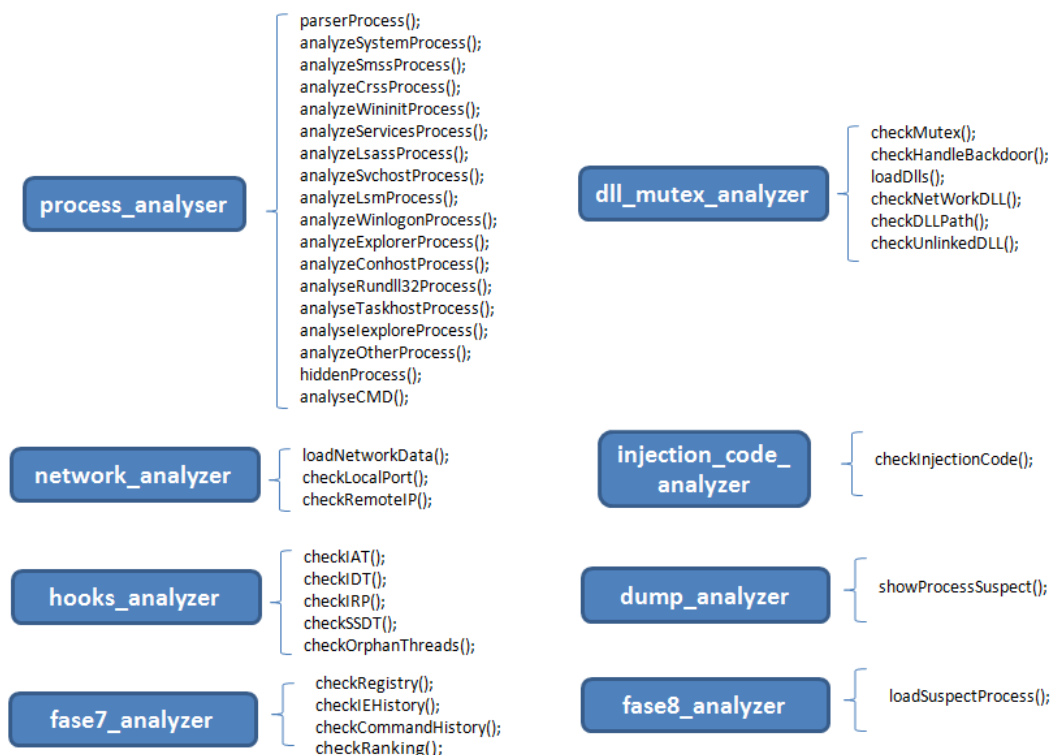


Figura 4.6: Módulos de controle de cada fase da metodologia *Maldetect* e suas principais funções.

associado a este comportamento recebe uma pontuação conforme mostra a tabela 4.9. Os valores das pontuações foram obtidas de forma heurística, e tem o objetivo de aplicar maior pontuação ao comportamento mais comum entre os códigos malicioso. Essa pontuação poderá ser readequada a medida que várias análises forem realizadas de forma a incorporar o conhecimento de uma mudança no comportamento dos *malwares*.

Tabela 4.9: Pontuação de anomalias detectadas pela *Maldetect Tool*

Anomalia	Pontuação
Caminho incorreto	1
Filho incorreto	1
PID incorreto	1
Base priority incorreto	1
PPID incorreto	2
Pai ou filho de um processo suspeito	4

Processo sendo executado a partir de uma pasta temporária	5
Processo que possui *mutex* suspeito	5
Processo que possui características de *backdoor*	5
Processo que carregou DLL fora de contexto	2
Processo que utiliza técnicas de ocultação	3
Processo filho do cmd.exe	2
Processo que possui área de memória com permissão de escrita e execução	1
Processo que carrega uma DLL suspeita	pontuação da DLL suspeita
Username incorreto	1
DLL em caminho incorreto	2
DLL não vinculada	2
Thread órfã	2
Realiza *Hook*	5
Processo em modo *debug*	2
Session running incorreto	1
Porta ou conexão suspeita	1
Processo com nome similar ao nome de um processo do Sistema Operacional	2
Detecção no Virustotal	Um ponto para cada antivírus que detectar o artefato como *malware*
Linha de comando do svchost.exe sem parâmentro -k	4

5 RESULTADOS E DISCUSSÕES

Como prova de conceito da metodologia *Maldetect* e validação da implementação realizada pela ferramenta *Maldetect Tool*, foram realizadas cinco análises de *dumps* de memória volátil infectados com códigos maliciosos diferentes. A seguir, será descrito o ambiente criado para captura dos *dumps* infectados e sua posterior análise pela *Maldetect Tool*.

5.1 Ambiente de Laboratório

Para realizar estas análises, foi criado um ambiente de laboratório com duas estações virtualizadas, utilizando a ferramenta *VirtualBox*, com as seguintes especificações:

* Uma máquina virtualizada com o Sistema Operacional Windows 7 Professional Service Pack 1, com 1GB de memória, 25GB de HD, uma interface de rede (modo rede interna) e uma CPU. Esta máquina não possui nenhum antivírus instalado nem acesso à Internet.
* Uma máquina virtualizada com o Sistema Operacional Linux Kali, com 2GB de memória, 64GB de HD, duas interfaces de rede (uma em modo rede interna e outra em modo bridge) e duas CPUs. Uma interface de rede para transferir arquivos da máquina Windows para o Linux e outra interface com conexão com a Internet. A *Maldetect Tool* está instalada nesta máquina virtual.

O distribuição Linux Kali foi escolhida porque possui várias ferramentas utilizadas em testes de penetração, análise forense de disco e de memória volátil. Ou seja, possui instalada as ferramentas necessárias para execução da *Maldetet Tool*, em especial o *Python* e o *Volatility*.

A figura 5.1 mostra a topologia da rede utilizada para realizar a captura e análise do *dump* de memória.

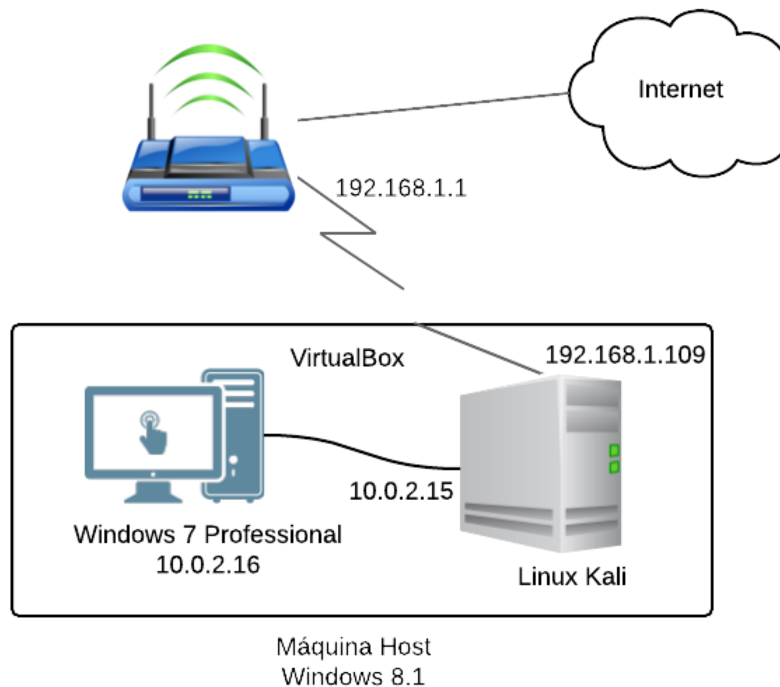


Figura 5.1: Topologia da rede do laboratório de análise automática de *dump* de memória.

5.2 Procedimentos

Ao final da instalação do Sistema Operacional Windows 7, criou-se um *snapshot* da máquina virtual livre de código malicioso no *VirtualBox*. Em seguida, foi executado cada *malware* listado na tabela 5.1 e realizou-se a captura do *dump* da memória da estação infectada com cada um dos códigos maliciosos utilizando a ferramenta Dumpit[7].

Os *dumps* de memória obtidos foram transferidos para a máquina com o Sistema Operacional Kali Linux onde foram analisados de forma automatizada utilizando a *Maldetect Tool*. A ferramenta não sabia previamente nenhuma informação sobre o artefato malicioso que havia nos *dumps* de memória.

Foi realizada uma análise automática de quatro *dumps* de memória de uma máquina Windows 7 infectada com os seguintes *malwares*: jackal[8], nfe.xml.exe[9], CiGPxdM.exe[10]

[7]Dumpit é uma ferramenta para aquisição de memória volátil do Sistema Operacional Windows. Esta ferramenta foi desenvolvida pela empresa MoonSols e pode ser obtida no seguinte endereço: http://www.moonsols.com/2011/07/18/moonsols-dumpit-goes-mainstream/

[8]md5: e0208ab8930434036cbeef5683418d23

[9]md5: f6be0475e183335e00ffe363cf62a2bc

[10]md5: 116addcf779c596ad11a3fe910050c9e

e NF-e 18454310845.exe[11].

Analisou-se um *malware* menos conhecido pelos antivírus como é o caso do jackal.exe. Inclusive, o antivírus Kaspersky não o detectou como sendo um código malicioso. A tabela 5.1 mostra a taxa de detecção desses *malwares* no *Virustotal*.

Tabela 5.1: Resultado da submissão dos códigos maliciosos ao VirusTotal.

Malwares	Taxa de detecção	Data da Análise
jackal.exe	20 / 57	26/04/2015 15:30:39 UTC
nfe.xml.exe	34 / 57	26/04/2015 15:34:43 UTC
NF-e 18454310845.exe	40 / 57	26/04/2015 15:40:05 UTC
CiGPxdM.exe	46 / 57	26/04/2015 15:37:12 UTC

Adicionalmente, realizou-se a análise de um *dump* de memória de Windows 7 64 bits (LIGH et al., 2014) por estar infectado com um *malware* que cria um processo oculto para verificar a eficácia de detecção da *Maldetect Tool*. A seguir serão mostrados os resultados dessas análises realizadas pela *Maldetect Tool*.

5.3 Resultados

Esta seção descreverá as anomalias detectadas durante a execução de cada fase da metodologia implementada pela *Maldetect Tool*.

5.3.1 Análise de Processos

A tabela 5.2 apresenta os processos suspeitos e as anomalias encontradas durante a análise do *dump* de memória volátil infectado com cada *malware*. Os processos suspeitos detectados nas análises dos *dumps* de memória infectado com os *malwares* CIGPxdM.exe e NF-e 18454310845.exe possuem nomes de processos legítimos do Sistema Operacional (mspaint.exe e svchost.exe), porém a localização no sistema de arquivo está incorreta. Na análise do *malware* jackal.exe, foi detectado como anomalia

[11]md5: f9856997401fd45a38790dcb1402537e

a criação de dois processos “filhos”, chamados de “*cmd.exe*”, pois o processo “*cmd.exe*” normalmente é filho do processo “*explorer.exe*”.

O processo suspeito que foi detectado na análise do *dump* de memória infectado com o *malware* “nfe.xml.exe” possui este nome (MALDETECT-PC.exe) pois era o *hostname* da máquina com o Sistema Operacional Windows 7, onde o código malicioso foi executado. E por fim, o processo detectado na análise do *Dump* infectado (LIGH et al., 2014) é um processo oculto, ou seja, não seria listado pelo gerenciador de processos do Sistema Operacional. Para coletar as informações deste processo foi necessário utilizar a opção –offset do *plugin procinfo*, conforme descrito no capítulo anterior.

Tabela 5.2: Atividades típicas de códigos maliciosos encontradas na fase de análise de processos e seus respectivos artefatos. O ranking é o somatório da pontuação das anomalias detectadas conforme a tabela 4.9

Malwares	Artefato	Ranking	Atividade maliciosa
jackal.exe	jackal.exe.exe	4	Cria dois processos cmd.exe!
nfe.xml.exe	MALDETECT-PC.exe	5	Este processo esta sendo executado a partir da pasta appData!
CiGPxdM.exe	svchost.exe	8	Pai não encontrado! Caminho incorreto! Username incorreto! Falta parâmetro -k!
CiGPxdM.exe	mspaint.exe	4	Filho de Processo suspeito!
Dump infectado	mswinnt.exe	12	Utiliza técnicas de ocultação de processos! Possui nome semelhante a um processo legítimo (wininit.exe)! Processo executado da pasta “users”

NF-e 18454310845.exe	svchost.exe	8	Pai não encontrado! Caminho incorreto! Username incorreto! Falta parâmetro -k!

5.3.2 Análise de bibliotecas dinâmicas e *handles*

A tabela 5.3 apresenta as anomalias dos artefatos suspeitos detectadas nesta fase. Nas análises dos *malwares* "CIGPxdM.exe", "jackal.exe" e "nfe.xml.exe", identificou-se DLLs carregadas fora de contexto, pois são DLLs utilizadas para realizar atividades de rede e os processos que as carregaram não são conhecidos como artefatos que, geralmente, usam este tipo de recurso. No caso da análise do código malicioso "CIGPxdM.exe", os processos de Calculadora (calc.exe) e Microsoft Paint (mspaint.exe), com certeza não deve fazer uso de recurso de rede e, portanto, não deveriam ter carregado DLLs neste contexto.

Apesar da análise do "NF-e 18454310845.exe" não ter detectado anomalia nesta fase, o processo suspeito "svchost.exe", detectado na fase anterior, carregou uma DLL de rede. Porém, ele possui nome de um processo que tipicamente utiliza recursos de rede.

Na análise do *Dump* infectado (LIGH et al., 2014), foi detectado, como anomalia, uma DLL em memória que estava localizada no sistema de arquivos na pasta "*users*".

Tabela 5.3: Atividades típicas de códigos maliciosos encontradas na fase de análise de bibliotecas dinâmicas e *handles* de processos. O ranking é o somatório da pontuação das anomalias detectadas conforme a tabela 4.9

Malwares	Artefato	Ranking	Atividade maliciosa
jackal.exe	jackal.exe.exe	16	Mutex malicioso encontrado: __Dassara__! Backdoor! Carrega três DLLs num contexto suspeito!

nfe.xml.exe	MALDETECT-PC.exe	2	Carrega uma DLL num contexto suspeito!
CiGPxdM.exe	mspaint.exe	6	Carrega três DLLs num contexto suspeito!
CiGPxdM.exe	calc.exe	6	Carrega três DLLs num contexto suspeito!
Dump infectado	encodedll.dll	7	DLL armazenada na pasta “users” DLL em caminho suspeito!
NF-e 18454310845.exe	—	0	Nenhuma atividade maliciosas detectada nesta fase!

5.3.3 Análise de Artefatos de Rede

A tabela 5.4 apresenta as anomalias dos artefatos suspeitos detectadas em cada uma das análises realizadas. Na análise do *malware* “jackal.exe”, o processo suspeito abriu a porta 9090 em modo *listening*, como esta porta não foi informada como sendo não nociva na fase pré-análise e não faz parte da lista de portas do Sistema Operacional, esta atividade de rede foi marcada como suspeita pela *Maldetect Tool*.

Já na análise do *Dump* infectado (LIGH et al., 2014), trata-se de conexões estabelecidas para IP’s de baixa reputação[12], porém sete das oito conexões detectadas estavam em estado de CLOSED e assim seus IP’s não estavam mais disponíveis em memória no momento da captura do *dump*, por isso são falsos positivos. Assim, apenas um IP suspeito possui um conexão em modo ESTABLISHED que é: 207.171.163.151 (detectado como suspeito por 2/40 *blacklist* pelo IPVoid).

Tabela 5.4: Atividades típicas de códigos maliciosos encontradas na fase de análise de artefatos de rede. O ranking é o somatório da pontuação das anomalias detectadas conforme a tabela 4.9

Malwares	Artefato	Ranking	Atividade maliciosa

[12]IP’s com histórico de conexões suspeitas de atividades maliciosas

jackal.exe	jackal.exe.exe	1	Porta ou conexão suspeita!
nfe.xml.exe	—	0	Nenhuma atividade maliciosas detectada nesta fase!
CiGPxdM.exe	—	0	Nenhuma atividade maliciosas detectada nesta fase!
Dump infectado	iexplorer.exe	8	Oito conexões suspeitas
NF-e 18454310845.exe	—	0	Nenhuma atividade maliciosas detectada nesta fase!

5.3.4 Busca por injeção de código

A tabela 5.5 apresentada as anomalias detectadas nesta fase. As anomalias identificadas nas análises dos *dumps* de memória infectados com os *malwares* "CIGPxdM.exe" e "NF-e 18454310845.exe", podem indicar que são processos ocos, ou seja, processos que podem ter sidos carregados de maneira legítima porém tiveram seus códigos substituídos em tempo de execução. Para confirmar este suspeita seria necessária uma comparação entre o conteúdo do processo em memória com o conteúdo do arquivo localizado no sistema de arquivos, porém esta comparação não faz parte da metodologia *Maldetect*, pois a metodologia tem como premissa apenas a análise do *dump* da memória volátil.

Tabela 5.5: Atividades típicas de códigos maliciosos encontradas na fase de busca por injeção de código. O ranking é o somatório da pontuação das anomalias detectadas conforme a tabela 4.9

Malwares	Artefato	Ranking	Atividade maliciosa
jackal.exe	—	0	Nenhuma anomalia detectada nesta fase!
nfe.xml.exe	MALDETECT-PC.exe	1	Possui área de memória com a flag de write_exec!

CiGPxdM.exe	svchost.exe	1	Possui área de memória com a flag de write_exec!
CiGPxdM.exe	mspaint.exe	1	Possui área de memória com a flag de write_exec!
CiGPxdM.exe	calc.exe	1	Possui área de memória com a flag de write_exec!
Dump infectado	explorer.exe	1	Possui área de memória com a flag de write_exec!
NF-e 18454310845.exe	svchost.exe	1	Possui área de memória com a flag de write_exec!

5.3.5 Busca por ***hooks***

A tabela 5.6 apresenta as anomalias detectadas nesta fase. Todos os *hooks* detectados nesta fase são em nível de usuário, mostrando que nenhum código malicioso conseguiu escalar privilégio para executar um *hook* em nível de Kernel. Outra fato interessante, é que todos burlaram as mesmas funções da DLL wow64.dll (Wow64PrepareForDebuguerAtach e Wow64SupendLocalthread). Estas funções atacadas permitem que o código malicioso coloque um processo em modo de depuração, como as ferramentas de *debuguer* realizam, e assim poderem pausar sua execução e alterar seus códigos e reiniciar sua execução.

Tabela 5.6: Atividades típicas de códigos maliciosos encontradas na fase de busca por *hooks*. O ranking é o somatório da pontuação das anomalias detectadas conforme a tabela 4.9

Malwares	Artefato	Ranking	Atividade maliciosa
jackal.exe	—	0	Nenhuma anomalia detectada nesta fase!
nfe.xml.exe	—	0	Nenhuma anomalia detectada nesta fase!

CiGPxdM.exe	svchost.exe	10	Processo realiza dois *hooks* suspeitos!
CiGPxdM.exe	mspaint.exe	10	Processo realiza dois *hooks* suspeitos!
CiGPxdM.exe	calc.exe	10	Processo realiza dois *hooks* suspeitos
Dump infectado	iexplorer.exe	10	Processo realiza dois *hooks* suspeitos
NF-e 18454310845.exe	svchost.exe	10	Processo realiza dois *hooks* suspeitos!

5.3.6 *Dump* de processos, bibliotecas e *drivers* suspeitos

A tabela 5.7 apresenta os resultados da consulta ao Virustotal utilizando *hashes sha-256* dos artefatos suspeitos. Os artefatos que foram detectados por algum antivírus receberam pontuação igual ao número de *hits* de detecção. É normal que alguns artefatos não sejam conhecidos pelo VirusTotal, visto que o processo reconstruído a partir do *dump* de memória sofre algumas alterações em relação ao arquivo em disco. Outro fato que contribui para não detecção do VirusTotal é que estes artefatos detectados durante a execução da *Maldetect Tool* pode não ser o mesmo código usado para disseminação do *malware.*

Tabela 5.7: Atividades típicas de códigos maliciosos encontradas na fase de *dump* de processos, bibliotecas e *drivers* suspeitos. O ranking é o somatório da pontuação das anomalias detectadas conforme a tabela 4.9

Malwares	Artefato	Ranking	Atividade maliciosa
jackal.exe	jackal.exe.exe	12	Taxa de Detecção do Virustotal: 12 / 57!
nfe.xml.exe	MALDETECT-PC.exe	0	Não encontrado no Virustotal!

CiGPxdM.exe	svchost.exe	0	Não encontrado no Virustotal!
CiGPxdM.exe	mspaint.exe	0	Taxa de Detecção do Virustotal: 0 / 57!
CiGPxdM.exe	calc.exe	0	Não encontrado no Virustotal!
Dump infectado	mswinnt.exe	2	Taxa de Detecção do Virustotal: 2 / 56!
Dump infectado	iexplorer.exe	0	Taxa de Detecção do Virustotal: 0 / 55!
Dump infectado	encodedll.dll	1	Taxa de Detecção do Virustotal: 1 / 56!
NF-e 18454310845.exe	svchost.exe	0	Não encontrado no Virustotal!

5.3.7 Processamento, Correlação e Relatório

Nesta fase, as pontuações de cada anomalia foram processadas e correlacionadas, gerando o *ranking* final de cada artefato. Também foi gerado um relatório em PDF de cada análise que está disponível no repositório do *GitHub* no seguinte endereço: *https://github.com/maldetect/*.

Além disso, foram coletadas as informações de histórico de comandos do prompt, histórico de navegação do Internet Explorer e o valor da chave de registro Microsoft\Windows\CurrentVersion\Run as quais estão disponíveis nos relatórios das análises.

5.3.8 Criação da base de conhecimento

Nesta fase, os artefatos que receberam alguma pontuação correspondente às anomalias detectadas nas fases anteriores, conforme tabela 4.9, são apresentados em ordem decrescente e permite as seguintes ações:

- GerarIOC: será gerado o XML contendo a lista de anomalias realizada por um determinado artefato, as quais foram detectadas durante a análise.

- CompararIOC: realiza uma busca no banco de conhecimento se o XML correspondente a um determinado artefato já está armazenado.

- SalvarIOC: armazena o XML no banco de dados de conhecimento.

5.4 Resumo das análises

A tabela 5.8 mostra os artefatos considerados maliciosos pela *Maldetect Tool* e as respectivas atividades típicas de *malwares* encontradas.

Tabela 5.8: Atividades típicas de códigos maliciosos encontrados e seus respectivos artefatos.

Malwares	Artefato	Ranking	Atividade maliciosa
jackal.exe	jackal.exe.exe	33	Cria dois processos cmd.exe! Mutex malicioso encontrado: __Dassara__! Porta ou conexão suspeita! Backdoor! Taxa de Detecção do Virustotal: 12 / 57! Carrega três DLLs num contexto suspeito!
nfe.xml.exe	MALDETECT-PC.exe	8	Este processo esta sendo executado a partir da pasta appData! Carrega uma DLL num contexto suspeito! Possui área de memória com a flag de write_exec!

CiGPxdM.exe	svchost.exe	19	Pai não encontrado! Caminho incorreto! Username incorreto! Falta parâmetro -k! Possui área de memória com a flag de write_exec! Processo realiza dois *hooks* suspeitos
CiGPxdM.exe	mspaint.exe	21	Filho de Processo suspeito! Possui área de memória com a flag de write_exec! Carrega três DLLs em contexto suspeito! Processo realiza dois *hooks* suspeitos
CiGPxdM.exe	calc.exe	17	Carrega três DLLs em contexto suspeito! Possui área de memória com a flag de write_exec! Processo realiza dois *hooks* suspeitos
Dump infectado	mswinnt.exe	22	Utiliza técnicas de ocultação de processos! Possui nome semelhante a um processo legítimo (wininit.exe)! Processo executado da pasta "users" Processo detectado por 2/56 no VirusTotal Carrega uma DLL que possui anomalias detectadas!
Dump infectado	iexplorer.exe	19	Possui área de memória com a flag de write_exec! Oito conexões suspeitas! Taxa de Detecção do Virustotal: 0 / 55!

Dump infectado	encodedll.dll	8	Processo executado da pasta "users" DLL em caminho suspeito! Taxa de Detecção do Virustotal: 1 / 56!
NF-e 18454310845.exe	svchost.exe	19	Pai não encontrado! Caminho incorreto! Username incorreto! Falta parâmetro -k! Possui área de memória com a flag de write_exec!

Percebeu-se que os *malwares* tentaram dificultar sua detecção usando nomes de processos correspondentes à nomes de processos legítimos do Sistema Operacional. Foram usados os seguintes nomes: svchost.exe (nome de processo que pertence ao núcleo do Sistema Operacional Windows 7), mspaint.exe (Microsoft Paint Brush) e calc.exe (Calculadora do Windows).

O *jackal.exe* foi executado a partir da pasta "c:\windows\system32" na tentativa de passar desapercebido como um processo legítimo do Windows 7.

Todos códigos maliciosos testados foram detectados. Além disso, a automatização proposta, implementada pela *Maldetect Tool*, reduziu consideravelmente o tempo de análise da memória volátil em relação a análise manual. A *Maldetect Tool* é capaz de identificar um *malware* desconhecido, pois detecta características comportamentais típicas de código malicioso. Ou seja, caso um *malware 0-day* utilize alguma das técnicas descrita neste trabalho, o mesmo seria identificado como um código indesejado e o relatório final apresentaria várias informações que auxiliariam na análise do *malware*.

Os relatórios gerado pela *Maldetect Tool* apresenta todas as informações relevantes coletadas durante a análise e direciona a atenção do analista para os artefatos que realmente realizam atividades típicas de *malware*, o que deixa claro qual foi o artefato malicioso encontrado.

Vale ressaltar que a *Maldetect Tool* não é uma ferramenta de análise de dinâmica de *malware*, mas sim uma ferramenta de detecção de códigos maliciosos baseada em comportamentos anômalos identificados através da análise do *dump* de memória volátil.

Por fim, na última fase da metodologia, um XML contendo a descrição das anomalias comportamentais detectadas pode ser gerado. Este arquivo possui os indicativos de comprometimentos (IOC) e pode ser salvo para popular a base conhecimento da *Maldetect Tool*. Ao final de cada análise, estes indicativos de comprometimentos podem ser comparados com a base de conhecimento para determinar se é uma artefato desconhecido ou se já existe um outro artefato que gerou as mesmas características comportamentais. A seguir um exemplo do XML gerado pela *Maldetect Tool* para a análise do malware NF-e 18454310845.exe que criou o artefato malicioso svchost.exe.

```
1  <maldetect>
2          <fase1>
3                  <svchost>
4                          <ppid>Não Encontrado</ppid>
5                          <path>c:\windows\SysWOW64\svchost.exe</path>
6                          <username>S
                               -1-5-21-855941912-591357546-2752705690-1000</
                               username>
7                          <command_line>"c:\windows\system32\svchost.exe"</
                               command_line>
8                  </svchost>
9          </fase1>
10         <fase2></fase2>
11         <fase3></fase3>
12         <fase4>
13                 <injection_code>
14                         <memory_write_exec>true</memory_write_exec>
15                 </injection_code>
16         </fase4>
17         <fase5></fase5>
18         <fase6></fase6>
19         </maldetect>
```

A *Maldetect Tool* ainda está em desenvolvimento, e está na versão *beta* para testes e melhorias das técnicas de detecção de anomalias comportamentais típicas de códigos maliciosos.

6 CONCLUSÕES E TRABALHOS FUTUROS

Neste trabalho, estudou-se três metodologias de análise de *dump* de memória volátil, as quais, embasaram a elaboração da *Maldetect*, que é uma metologia automatizável para detecção de *malwares* desconhecidos. A *Maldetect* coleta e correlaciona informações comportamentais dos processos, DLLs e *drivers* de um *dump* de memória volátil (RAM) e identifica quais desses comportamentos são típicos de códigos maliciosos. A metodologia pode ser utilizada para detectar as ameaças que são consideradas não-convencionais, tais como, *malwares* que exploram vulnerabilidades *0-day* e *malwares* desconhecidos.

Com base nesta metodologia, foi construída uma ferramenta utilizando as linguagens *Python* e PHP, denominada *Maldetect Tool*, a qual implementa a metodologia *Maldetect*. A ferramenta utiliza recursos do *Volatility* para extrair informações do *dump* de memória volátil. E, para coletar os dados dos processos em execução na memória, no momento da aquisição do *dump*, foi construído o *plugin procinfo* para o *Volatility*.

Além disso, a *Maldetect Tool* detectou *malwares* com baixa taxa de detecção no *VirusTotal*. Essa detecção foi baseada na identificação de anomalias comportamentais. Assim, demonstrou-se que este tipo de detecção é mais eficaz que a detecção baseada em assinatura utilizada pela maioria dos antivírus, já que pequenas modificações no código do *malware* podem alterar sua assinatura sem alterar seu comportamento malicioso.

Assim, apesar do aumento da complexidade e do avanço das técnicas utilizadas pelos *malwares* modernos, coletar e correlacionar informações comportamentais de várias fontes é uma das maneiras eficientes de detectá-los.

A *Maldetect Tool* suporta a incorporação de novas técnicas de detecção e assim pode-se utilizar um ciclo de aperfeiçoamento, tal como o PDCA (PLAN-DO-CHECK-ACT) para realizar melhorias na metodologia e na ferramente construída. Ou seja, a partir da análise de um novo *malware*, que utilizou uma técnica não detectada pela ferramenta proposta ou não prevista na metodologia, este conhecimento pode ser incorporado à *Maldetect Tool* e novos *malwares* que utilizem a mesma técnica serão detectados a partir de então.

A solução apresentada neste projeto não substitui as ferramentas tradicionais de detecção de artefatos maliciosos, pois detecta códigos indesejados que burlaram as ferramentas tradicionais e infectaram uma ou mais estações. Dessa forma, a *Maldetect Tool* trata o risco residual deixado pelas soluções já existentes. Além disso, seguindo uma metodologia de análise e automatizando suas tarefas, ocorre uma redução considerável no tempo da análise da memória volátil. Isso permite o monitoramento e a varredura de vários computadores de uma rede em intervalos de tempos curtos. Logo, o tempo de detecção de um ataque que burlou os mecanismos tradicionais também serão reduzidos, o que pode impedir o "espalhamento" para outras estações.

Em trabalhos futuros, sugere-se a implementação para outros Sistemas Operacionais e a ampliação da base de conhecimento dos indicativos de comprometimentos dos vários tipos de códigos malicioso, com o objetivo de aplicar técnicas de aprendizado de máquina. E, assim, agrupar os *malwares* mediante algumas características intrínsecas, a fim de ajudar na mitigação e no combate aos códigos maliciosos ainda não-conhecidos.

REFERÊNCIAS BIBLIOGRÁFICAS

AJJAN, A. *Ransomware: Next-Generation Fake Antivirus*. 2013. A SophosLabs technical paper. http://www.sophos.com/en-us/medialibrary/PDFs/technicalpapers/SophosRansomwareFakeAntivirus.pdf?la=en.pdf?dl=true.

AV-TEST. *Malware Statistics*. 2015. AV-TEST The Independent IT-Security Institute. https://www.av-test.org/en/statistics/malware/.

BAILEY, M. et al. Automated classification and analysis of internet malware. In: SPRINGER. *Recent advances in intrusion detection*. [S.l.], 2007. p. 178–197.

BLUNDEN, B. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. [S.l.]: Jones & Bartlett Publishers, 2011.

COIMBRA, J. F. M. Estudo da vulnerabilidade de heap overflow e medidas de proteção. 2011.

DATA, G. *Uroburos - Highly complex espionage software with Russian roots*. 2014. G Data SecurityLabs. https://public.gdatasoftware.com/Web/Content/INT/Blog/2014/02_2014/documents/GData_Uroburos_RedPaper_EN_v1.pdf.

DFRWS. *The DFRWS 2005 forensic challenge*. 2005. http://www.dfrws.org/2005/challenge/index.shtml.

ESHAN, F. *Memory Forensics & Security Analytics: Detecting Unknown Malware*. 2014. http://www.isaca.org/chapters5/Ireland/Documents/2014EventPresentations/DetectingUnknownMalwareMemoryForensicsandSecurityAnalytics-FahadEhsan.pdf.

FOUNDATION, V. *Command Reference*. 2015. https://github.com/volatilityfoundation/volatility/wiki/Command-Reference.

HU, L. et al. Analyzing malware based on volatile memory. *Journal of Networks*, v. 8, n. 11, p. 2512–2519, 2013.

HYDE, R. *The art of assembly language*. [S.l.]: No Starch Press, 2010.

LEE, R. *Finding Unknown Malware Step By Step*. 2013. SANS DFIR Faculty. http://digital-forensics.sans.org/media/poster_fall_2013_forensics_final.pdf.

LI, F. A detailed analysis of an advanced persistent threat malware. *SANS Institute InfoSec Reading Room*, 2011.

LIGH, M. et al. *Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code*. [S.l.]: Wiley Publishing, 2010.

LIGH, M. H. et al. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. [S.l.]: John Wiley & Sons, 2014.

LOCK, H.-Y. *Using IOC (Indicators of Compromise) in Malware Forensics*. 2013. http://www.sans.org/reading-room/whitepapers/forensics/ioc-indicators-compromise-malware-forensics-34200.

LYDA, R.; HAMROCK, J. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, IEEE, n. 2, p. 40–45, 2007.

MACAFEE. *Protecting Your Critical Assets*. 2010. McAfee Labs and McAfee Foundstone Professional Services. http://www.wired.com/images_blogs/threatlevel/2010/03/operationaurora_wp_0310_fnl.pdf.

MALIN, C. H.; CASEY, E.; AQUILINA, J. M. *Malware forensics: investigating and analyzing malicious code*. [S.l.]: Syngress, 2008.

MANDIANT. *Redline Users Guide*. 2012. Mandiant Corporation. https://dl.mandiant.com/EE/library/Redline1.7_UserGuide.pdf.

MICROSOFT. *O que é uma DLL?* 2014. Suporte MSDN. https://support.microsoft.com/pt-br/kb/815065.

OLSEN, P. *Know your Windows Processes or Die Trying*. 2014. Sysforensics. http://sysforensics.org/2014/01/know-your-windows-processes/.

OROSZLANY, M. *Rootkits under Windows OS and methods of their detection*. 2008. Masaryk University Faculty of Informatics. http://is.muni.cz/th/139801/fi_b/Bc.pdf.

RUSSINOVICH, M.; SOLOMON, D. A.; IONESCU, A. *Windows Internals, Part 1. 6th Edition*. [S.l.]: Microsoft Press, 2012.

SANGER, D. E. *Obama Order Sped Up Wave of Cyberattacks Against Iran*. 2012. New York Times. http://www.nytimes.com/2012/06/01/world/middleeast/obama-ordered-wave-of-cyberattacks-against-iran.html?pagewanted=all&_r=1.

SANS. *Know Normal... Find Evil*. 2014. SANS DFIR. https://digital-forensics.sans.org/media/poster_2014_find_evil.pdf.

SEGER, R. *Hunting the Mutex*. 2014. http://researchcenter.paloaltonetworks.com/2014/08/hunting-mutex/.

STÜTTGEN, J.; COHEN, M. Anti-forensic resilient memory acquisition. *Digital Investigation*, Elsevier, v. 10, p. S105–S115, 2013.

TILBURY, C. *Memory Forensics*. 2012. SANS Computer Forensics and Incident Response. http://software.msu.montana.edu/free/ISSA/MemoryForensicsMadeEasySolvingCaseswiththeNewBreedofTools-Tilbury-5-22-2012.pdf.

VÖMEL, S.; FREILING, F. C. A survey of main memory acquisition and analysis techniques for the windows operating system. *Digital Investigation*, Elsevier, v. 8, n. 1, p. 3–22, 2011.

WOODHULL, A.; TANENBAUM, A. As sistemas operacionais: Projeto e implementação. *Porto Alegre*, 2000.

YANG, G.; TIAN, Z.; DUAN, W. The prevent of advanced persistent threat. *Journal of Chemical and Pharmaceutical Research*, v. 6, n. 7, p. 572–576, 2014.

ZHANG, B.; YIN, J.; HAO, J. Using fuzzy pattern recognition to detect unknown malicious executables code. In: *Fuzzy Systems and Knowledge Discovery*. [S.l.]: Springer, 2005. p. 629–634.

APÊNDICES

A Código do *plugin*: procinfo

```
1  import volatility.utils as utils
2  import volatility.commands as commands
3  import volatility.win32.tasks as tasks
4  import re
5  import volatility.obj as obj
6  import volatility.plugins.taskmods as taskmods
7
8  """Trecho de código retirado do plugin getsids"""
9  def find_sid_re(sid_string, sid_re_list):
10     for reg, name in sid_re_list:
11         if reg.search(sid_string):
12             return name
13
14 well_known_sid_re = [
15   (re.compile(r'S-1-5-[0-9-]+-500'), 'Administrator'),
16   (re.compile(r'S-1-5-[0-9-]+-501'), 'Guest'),
17   (re.compile(r'S-1-5-[0-9-]+-502'), 'KRBTGT'),
18   (re.compile(r'S-1-5-[0-9-]+-512'), 'Domain Admins'),
19   (re.compile(r'S-1-5-[0-9-]+-513'), 'Domain Users'),
20   (re.compile(r'S-1-5-[0-9-]+-514'), 'Domain Guests'),
21   (re.compile(r'S-1-5-[0-9-]+-515'), 'Domain Computers'),
22   (re.compile(r'S-1-5-[0-9-]+-516'), 'Domain Controllers'),
23   (re.compile(r'S-1-5-[0-9-]+-517'), 'Cert Publishers'),
24   (re.compile(r'S-1-5-[0-9-]+-520'), 'Group Policy Creator Owners'),
25   (re.compile(r'S-1-5-[0-9-]+-533'), 'RAS and IAS Servers'),
26   (re.compile(r'S-1-5-5-[0-9]+-[0-9]+'), 'Logon Session'),
27   (re.compile(r'S-1-5-21-[0-9-]+-518'), 'Schema Admins'),
28   (re.compile(r'S-1-5-21-[0-9-]+-519'), 'Enterprise Admins'),
29   (re.compile(r'S-1-5-21-[0-9-]+-553'), 'RAS Servers'),
30 ]
31
32 well_known_sids = {
33   'S-1-0': 'Null Authority',
34   'S-1-0-0': 'Nobody',
35   'S-1-1': 'World Authority',
36   'S-1-1-0': 'Everyone',
37   'S-1-2': 'Local Authority',
38   'S-1-2-0': 'Local (Users with the ability to log in locally)',
39   'S-1-2-1': 'Console Logon (Users who are logged onto the physical
       console)',
```

```
40     'S-1-3': 'Creator Authority',
41     'S-1-3-0': 'Creator Owner',
42     'S-1-3-1': 'Creator Group',
43     'S-1-3-2': 'Creator Owner Server',
44     'S-1-3-3': 'Creator Group Server',
45     'S-1-3-4': 'Owner Rights',
46     'S-1-4': 'Non-unique Authority',
47     'S-1-5': 'NT Authority',
48     'S-1-5-1': 'Dialup',
49     'S-1-5-2': 'Network',
50     'S-1-5-3': 'Batch',
51     'S-1-5-4': 'Interactive',
52     'S-1-5-6': 'Service',
53     'S-1-5-7': 'Anonymous',
54     'S-1-5-8': 'Proxy',
55     'S-1-5-9': 'Enterprise Domain Controllers',
56     'S-1-5-10': 'Principal Self',
57     'S-1-5-11': 'Authenticated Users',
58     'S-1-5-12': 'Restricted Code',
59     'S-1-5-13': 'Terminal Server Users',
60     'S-1-5-14': 'Remote Interactive Logon',
61     'S-1-5-15': 'This Organization',
62     'S-1-5-17': 'This Organization (Used by the default IIS user)',
63     'S-1-5-18': 'Local System',
64     'S-1-5-19': 'NT Authority/Local Service',
65     'S-1-5-20': 'NT Authority/Network Service',
66     'S-1-5-32-544': 'Administrators',
67     'S-1-5-32-545': 'Users',
68     'S-1-5-32-546': 'Guests',
69     'S-1-5-32-547': 'Power Users',
70     'S-1-5-32-548': 'Account Operators',
71     'S-1-5-32-549': 'Server Operators',
72     'S-1-5-32-550': 'Print Operators',
73     'S-1-5-32-551': 'Backup Operators',
74     'S-1-5-32-552': 'Replicators',
75     'S-1-5-32-554': 'BUILTIN\Pre-Windows 2000 Compatible Access',
76     'S-1-5-32-555': 'BUILTIN\Remote Desktop Users',
77     'S-1-5-32-556': 'BUILTIN\Network Configuration Operators',
78     'S-1-5-32-557': 'BUILTIN\Incoming Forest Trust Builders',
79     'S-1-5-32-558': 'BUILTIN\Performance Monitor Users',
80     'S-1-5-32-559': 'BUILTIN\Performance Log Users',
81     'S-1-5-32-560': 'BUILTIN\Windows Authorization Access Group',
82     'S-1-5-32-561': 'BUILTIN\Terminal Server License Servers',
83     'S-1-5-32-562': 'BUILTIN\Distributed COM Users',
84     'S-1-5-32-568': 'BUILTIN\IIS IUSRS',
```

```
85   'S-1-5-32-569': 'Cryptographic Operators',
86   'S-1-5-32-573': 'BUILTIN\Event Log Readers',
87   'S-1-5-32-574': 'BUILTIN\Certificate Service DCOM Access',
88   'S-1-5-33': 'Write Restricted',
89   'S-1-5-64-10': 'NTLM Authentication',
90   'S-1-5-64-14': 'SChannel Authentication',
91   'S-1-5-64-21': 'Digest Authentication',
92   'S-1-5-80': 'NT Service',
93   'S-1-5-86-1544737700-199408000-2549878335-3519669259-381336952': 'WMI (
        Local Service)',
94   'S-1-5-86-615999462-62705297-2911207457-59056572-3668589837': 'WMI (
        Network Service)',
95   'S-1-5-1000': 'Other Organization',
96   'S-1-16-0': 'Untrusted Mandatory Level',
97   'S-1-16-4096': 'Low Mandatory Level',
98   'S-1-16-8192': 'Medium Mandatory Level',
99   'S-1-16-8448': 'Medium Plus Mandatory Level',
100  'S-1-16-12288': 'High Mandatory Level',
101  'S-1-16-16384': 'System Mandatory Level',
102  'S-1-16-20480': 'Protected Process Mandatory Level',
103  'S-1-16-28672': 'Secure Process Mandatory Level',
104 }
105
106 """Fim do trecho de código retirado do plugin getsids"""
107
108 class ProcInfo (commands.Command):
109
110         def __init__(self, config, *args):
111                 """Este função registra o parâmetro -o na linha de
                    comando para que o offset seja informado
112                         Uso do plugin sem offset: vol -f [dump] --profile
                            =[profile] procinfo
113                         Uso do plugin com offset: vol -f [dump] --profile
                            =[profile] procinfo --offset=[offset]
114                 """
115                 commands.Command.__init__(self, config, *args)
116                 self._config.add_option('offset', short_option='o',
                    default=None,
117                                 help='Physical Address', type='int')
118
119
120         def calculate (self):
121                 """Esta função obtém um ponteiro para lista de processos,
                        e no caso do offset ser informado obtém um ponteiro
                    para estrutura de memória do processo"""
```

```
122             config= self._config
123             addr_space = utils.load_as(self._config)
124             if (config.offset != None):     """Caso o offset seja
                    infomado"""
125                 offset = taskmods.DllList.
                        virtual_process_from_physical_offset(
                        addr_space, config.offset).obj_offset
126                 yield obj.Object("_EPROCESS", offset = offset, vm
                        = addr_space)
127             else:   """Caso o offset nÃ£o seja informado"""
128                 for proc in tasks.pslist(addr_space):
129                     yield proc
130 
131         def render_text(self, outfd, data):
132             """Esta função imprimi o resultado"""
133             outfd.write("PID ;| PPID ;| PROCESS_NAME ;| BASEPRIORITY
                    ;| PATH ;| COMMAND_LINE ;| SESSIONID ;| CREATE_TIME ;|
                     EXIT_TIME ;| HANDLES ;| THREADS ;| USERNAME\n")
134             for proc in data:
135                 token = proc.get_token()
136                 sids = list(token.get_sids())
137                 sid_string = sids[0]
138                 if sid_string in well_known_sids:
139                     sid_name = " ({0})".format(well_known_sids[
                            sid_string])
140                 else:
141                     sid_name_re = find_sid_re(sid_string,
                            well_known_sid_re)
142                     if sid_name_re:
143                         sid_name = " ({0})".format(sid_name_re)
144                     else:
145                         sid_name = ""
146 
147                 outfd.write("{0} ;| {1} ;| {2} ;| {3} ;| {4} ;|
                    {5} ;| {6} ;| {7} ;| {8} ;| {9} ;| {10} ;|
                    {11}\n".format(proc.UniqueProcessId, proc.
                    InheritedFromUniqueProcessId, proc.
                    ImageFileName,proc.Pcb.BasePriority,proc.Peb.
                    ProcessParameters.ImagePathName, proc.Peb.
                    ProcessParameters.CommandLine, proc.Peb.
                    SessionId,proc.CreateTime, proc.ExitTime, proc
                    .ObjectTable.HandleCount, proc.ActiveThreads,
                    sid_string + sid_name))
148 
149 
```

```
150
151 }
```

B Relatório do *malware* NF-e 18454310845.exe gerado pela ferramenta

Maldetect: Detecting Unknown Malware

Processos Maliciosos Encontrados:

PID	process_name	ranking	anomaly_description	log_information
1476	svchost.exe	19	Pai não encontrado! Caminho incorreto! username incorreto Falta parâmetro -k Área de memória com flag de write_exec! Processo realiza um hook suspeito na função: wow64.dll!Wow64PrepareForDebuggerAttach at 0x74c3d438! Processo realiza um hook suspeito na função: wow64.dll!Wow64SuspendLocalThread at 0x74c3c174!	Hash sha256: 053b6d67b0d4e2702ce466fef9d61fa60bff0657ecc6e777409d2d6264cb4c42, nÃ£o encontrado na base do virustotal!
1344	DumpIt.exe	15	Este artefato esta sendo executado a partir da pasta users! Processo realiza um hook suspeito na função: wow64.dll!Wow64PrepareForDebuggerAttach at 0x74c3d438! Processo realiza um hook suspeito na função: wow64.dll!Wow64SuspendLocalThread at 0x74c3c174!	
1928	explorer.exe	3	Porta ou conexão suspeita! Porta ou conexão suspeita! Área de memória com flag de write_exec!	
252	svchost.exe	2	Porta ou conexão suspeita! Porta ou conexão suspeita!	
1800	svchost.exe	1	Área de memória com flag de write_exec!	
1548	msiexec.exe	1	Área de memória com flag de write_exec!	
2404	mscorsvw.exe	1	Área de memória com flag de write_exec!	
2628	mscorsvw.exe	1	Área de memória com flag de write_exec!	

472	conhost.exe	1	Caminho incorreto!	

DLLs Maliciosas Encontrados:

base	DLL	ranking	anomaly_description

Atividades de Rede Suspeitas:

PID	process_name	protocol	local_ip	local_port	remote_ip	remote_port	state
252	svchost.exe	TCPv4	-	49268	127.0.0.1	80	CLOSED
1928	explorer.exe	TCPv4	-	49267	127.0.0.1	80	CLOSED
252	svchost.exe	TCPv4	-	49264	224.0.0.22	80	CLOSED
1928	explorer.exe	TCPv4	-	49266	255.255.255.255	80	CLOSED

Histórico de Acessos do iexplore.exe:

pid	url

Histórico de Comando do cmd.exe:

****

CommandProcess: conhost.exe Pid: 644

CommandHistory: 0x356650 Application: cmd.exe Flags: Allocated, Reset

CommandCount: 1 LastAdded: 0 LastDisplayed: 0

FirstCommand: 0 CommandCountMax: 50

ProcessHandle: 0x60

Cmd #0 @ 0x3551c0: ipconfig

Cmd #15 @ 0x300158: 6

Cmd #16 @ 0x355950: 5

****

CommandProcess: conhost.exe Pid: 2636

CommandHistory: 0xb6700 Application: DumpIt.exe Flags: Allocated

CommandCount: 0 LastAdded: -1 LastDisplayed: -1

FirstCommand: 0 CommandCountMax: 50

ProcessHandle: 0x60

Cmd #15 @ 0x60158:

Cmd #16 @ 0xb5030: