Capítulo 3

Introdução à Análise de Códigos Maliciosos para ambiente Windows

Renato Marinho[1,2], Mateus Santos[1], Raimir Holanda[1,2], Antonio Horta[1,3]

[1]Morphus Segurança da Informação
[2]Universidade de Fortaleza (UNIFOR)
[3]Instituto Militar de Engenharia (IME)

Abstract

Malicious code analysis aims at a deep understanding of how malware works, what tactics, techniques and procedures are used, what tools are applied, if there are network operations, file exfiltration, capture of user information or the system, among other activities. In this mini-course, participants will be trained in malicious code analysis techniques for the Windows environment. The course applies a methodology that seeks, incrementally, to aggregate new information about the malware at each stage, comprising the following stages: binary profiling (OSINT), fully automated analysis, binary properties analysis, behavior analysis and manual reverse analysis.

Resumo

A análise de código malicioso visa o entendimento profundo do funcionamento de um malware, como ele atua, quais as táticas, técnicas e procedimentos são utilizados, quais ferramentas são empregadas, se há operações de rede, exfiltração de arquivos, captura de informações do usuário ou do sistema, entre outras atividades. Neste minicurso os participantes serão capacitados nas técnicas de análise de códigos maliciosos para o ambiente Windows. O curso aplica uma metodologia, que busca, de forma incremental, agregar novas informações sobre o malware em cada estágio, compreendendo os seguintes estágios: profiling do binário (OSINT), análise totalmente automatizada, análise de propriedades do binário, análise de comportamento e análise reversa manual.

© 2022 Sociedade Brasileira de Computação

3.1. Introdução

O número de ataques cibernéticos está, sem dúvida, em ascensão, tendo como alvos setores público e privado e tendo como um dos principais vetores de ataque códigos maliciosos (*malicious software* ou *malware*), que são arquivos executáveis que apresentam deliberadamente intenções prejudiciais à sistemas computacionais [Skoudis and Zeltser 2003], o que pode incluir do vazamento de informações sensíveis do usuário ao controle total do sistema infectado.

Esses ataques cibernéticos se concentram em indivíduos ou organizações com um esforço para extrair informações valiosas. Em alguns casos, esses ataques cibernéticos estão supostamente vinculados a crimes cibernéticos ou grupos patrocinados por países, mas também podem ser realizados por grupos individuais para atingir objetivos financeiros. Apesar do crescente investimento em tecnologias de segurança, ainda estamos vendo sérias violações de segurança acontecendo em grandes corporações e governos [Kim 2018].

Em sua quase totalidade, esses ataques cibernéticos usam software malicioso (também chamado de malware) para infectar seus alvos. Mais especificamente, um malware é um código que executa ações maliciosas, podendo assumir a forma de um executável, script, código ou qualquer outro software. Apesar do avanço no desenvolvimento de anti-malware, o número de ataques de malware está em tendência de alta [Sahay et al. 2020].

Em sua essência, o objetivo da análise de malware é geralmente fornecer as informações que você precisa para responder a uma intrusão de rede. Os objetivos são tipicamente determinar exatamente o que aconteceu, e garantir que todas as máquinas e arquivos infectados foram localizados [Sikorski 2012]. Ao analisar um malware suspeito, o objetivo normalmente será determinar, por exemplo, exatamente o que um binário suspeito em particular pode fazer, como detectá-lo na rede e como medir e conter seus danos.

Existem quatro abordagens fundamentais para a análise de malware: estática, dinâmica, código e memória [Monnappa 2018], [Kleymenov and Thabet 2019]. A análise estática consiste em um processo de analisar um binário sem executá-lo. Este tipo de análise pode confirmar se um arquivo é malicioso, fornecer informações sobre sua funcionalidade e, às vezes, fornecer informações que lhe permitirão produzir assinaturas de rede simples. A análise estática é direta e pode ser rápida, mas pode se demonstrar ineficaz contra malware sofisticado pois pode perder comportamentos importantes.

A análise dinâmica é o processo de execução do binário suspeito em um ambiente isolado e monitorando seu comportamento. Esta técnica de análise é fácil de executar e fornece informações valiosas sobre a atividade do binário durante sua execução. Entretanto, antes de executar o malware com segurança, deve-se configurar um ambiente cuidadosamente, de maneira que lhe permita estudar a execução do malware sem risco de danos ao sistema ou rede.

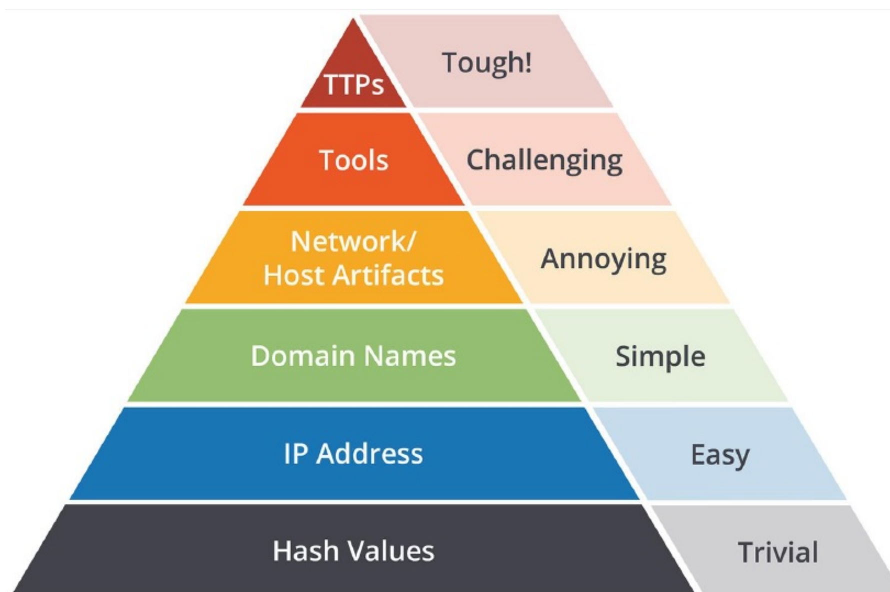
Análise de código é uma técnica que se concentra em analisar o código para entender o funcionamento interno do binário. Esta técnica revela informações que não são possíveis de determinar apenas a partir de análises estáticas ou dinâmicas. Esta análise é dividida em análise de código estática e análise de código dinâmica. A análise de código estática envolve desmontar o binário suspeito e olhar para o código para entender

o comportamento do programa, enquanto a análise de código dinâmica envolve depurar o binário suspeito de maneira controlada para entender sua funcionalidade. Nesta classe de análise é necessária uma compreensão de conceitos de linguagem de programação e sistema operacional.

A técnica de análise de memória consiste em analisar a memória RAM do computador para artefatos forenses. É tipicamente uma técnica forense, mas integrá-lo em uma análise de malware ajudará a obter uma compreensão do comportamento do malware após a infecção. A análise de memória é especialmente útil para determinar os recursos furtivos e evasivos do malware.

À medida que o malware se ofusca cada vez mais e aplica técnicas de anti-análise para impedir nossa análise, precisamos de métodos mais sofisticados que nos permitam levantar a cortina escura projetada para nos manter fora [Andriesse 2018]. De acordo com o tipo de análise realizada em um malware pode-se causar diferentes impactos para o atacante, o que costumamos chamar de Pirâmide da Dor [Bianco 2022]. Na Figura 3.1 observamos que análises triviais como a descoberta do *hash* de um malware causa um impacto mínimo, entretanto a descoberta das táticas, técnicas e procedimentos utilizados pelo malware tem um potencial de gerar um grande impacto para o atacante.

Figura 3.1. Pirâmide da Dor



Para o processo de análise de malware, os autores utilizaram uma metodologia, que busca, de forma incremental, agregar novas informações sobre o malware em cada estágio, conforme descrito na Figura 3.2. Como pode ser observado, a metodologia compreende os seguintes estágios: *profiling* do binário (OSINT), Análise totalmente automatizada, análise de propriedades do binário, análise de comportamento e análise reversa manual. Na aplicação desta metodologia e com o propósito de melhor exemplificar o uso de técnicas de análise, foi utilizado o malware TextInputDocsx.exe. Este artefato é classificado como um malware bancário e será utilizado na evolução das análises.

Figura 3.2. Metodologia do Processo de Análise de Malware. Fonte:Alissa Torres

Para fazer frente a este cenário atual de contínuas ameaças, precisamos de profissionais de cibersegurança que sejam pelo menos igualmente conhecedores daqueles que cometem esses crimes online. Ser capaz de desenvolver mecanismos adequados de detecção depende da compreensão como esses criminosos operam, como eles entram, o que procuram e como eles exfiltram o que encontraram da infraestrutura [Carvey 2014]. Nas demais seções deste capítulo são abordadas em detalhe a preparação de um laboratório e os estágios da metodologia do processo de análise de malware proposta pelos autores.

3.2. Preparação de um laboratório de análise

A utilização de um ambiente de laboratório devidamente preparado e isolado é um pré-requisito fundamental para análise de artefatos maliciosos. Este tipo de infraestrutura permite a investigação do artefato sem a exposição desnecessária do host do analista e da rede na qual se encontra a riscos de contaminação e prejuízos imprevisíveis. Ao analisar um código de um ransomware no seu próprio host, por exemplo, o analista corre o risco de executá-lo por acidente e ter sua estação de trabalho criptografada - sem contar com o risco de contaminação ou criptografia de outros hosts ou diretórios compartilhados conectados ao host.

Nesta seção, trataremos dos tipos de ambientes de laboratório, suas diferenças, vantagens e desvantagens e detalharemos como os alunos do minicurso devem proceder para a utilização de ambiente virtualizado para a realização das práticas.

3.2.1. Tipos de ambientes de análise

Ambientes de laboratório para análise de malware podem ser compostos por hosts físicos e/ou virtuais desconectados da rede e, de preferência, desconectados da internet. A recomendação de estarem desconectados da rede está ligado a prevenção de contaminação do ambiente e o isolamento da internet tem a ver com o maior controle do processo de análise e do entendimento do comportamento do artefato, conforme veremos mais adiante.

Quanto a questão do sistema ser físico ou virtual, há alguns pontos a serem con-

siderados. Alguns artefatos maliciosos podem fazer verificações do sistema operacional onde está sendo executado. Caso identifique que está em um ambiente virtual, podem não se manifestar ou se manifestar de maneira diferente para despistar o analista. Por outro lado, o ambiente físico apresenta desafios. Um deles é a remoção do malware após a infecção, que pode ser bastante trabalhosa e demorada de ser feita manualmente.Uma alternativa seria a utilização de ferramentas de recuperação da imagem completa do sistema após a infecção. Apesar de facilitar o processo, pode se tornar onerosa do ponto de vista de tempo.

Por estes motivos, é mais comum que o laboratório de análise seja composto por hosts virtuais. Nestes, é possível utilizar o recurso de *snapshot*, o que facilita o processo de execução do malware e de recuperação do estado do sistema de forma bastante rápida. Pesa contra este tipo de ambiente, no entanto, a questão de os malwares poderem "perceber"que estão sendo analisados em um ambiente controlado. Caberá ao analista identificar tais tentativas de identificação e contorná-las de alguma forma.

Por exemplo, o malware poderá tentar identificar hosts virtualizados por meio da leitura de chaves de registro do Windows que estão presentes somente nesta classe de hosts. As chaves de registro 'HKCU\SOFTWARE\VMware, Inc.\VMware Tools' e 'HARDWARE\ACPI \FADT\VBOX__', por exemplo, só estarão presentes em sistemas virtualizados com o VMWare e VirtualBox, respectivamente. Caso o analista perceba a leitura destas chaves e logo após uma decisão de interromper a execução do código, o analista poderá intervir no código invertendo a lógica de uma condicional 'jmp', por exemplo. A Figura 3.3 mostra um exemplo de código realizando a checagem da existência da chave do VirtualBox. Este e outros exemplos de técnicas de evasão podem ser encontrados nos projetos Evasion Techniques[1] do Checkpoint Research e Pafish[2] do pesquisador Alberto Ortega.

3.2.2. Ambiente do minicurso

Para este minicurso, utilizaremos um ambiente composto por dois hosts virtuais, sendo um Windows 10 e outro Linux. No sistema Windows, serão utilizadas ferramentas de análise estática e dinâmica de códigos maliciosos. O objetivo é utilizar o sistema, sobretudo, para a execução e o monitoramento do comportamento do artefato suspeito utilizando ferramentas tais como Process Hacker e Process Monitor. Este monitoramento pode propiciar a identificação, por exemplo, da criação de arquivos ou de mecanismos de persistência.

Já o sistema Linux será utilizado para a simulação de um ambiente de rede com o objetivo de analisar o comportamento de comunicação do artefato com hosts externos. É bastante comum que os códigos maliciosos estabeleçam conexões com hosts externos para download de novos componentes, para a exfiltração de dados ou somente para avisar o atacante que mais uma vítima foi contaminada.

Tanto a criação de arquivos como a tentativa de comunicação externa pelo malware podem ser úteis para o estabelecimento dos indicadores de comprometimento (IOCs) do artefato analisado. Estes IOCs são de grande valia na identificação do escopo de um incidente pois viabilizam a identificação de outros hosts contaminados.

[1] https://evasions.checkpoint.com/
[2] https://github.com/a0rtega/pafish

Figura 3.3. Exemplo de código malicioso buscando pela chave de registro do VirtualBox

```
/* sample of usage: see detection of VirtualBox in the table below to check registry path */
int vbox_reg_key7() {
    return pafish_exists_regkey(HKEY_LOCAL_MACHINE, "HARDWARE\\ACPI\\FADT\\VBOX__");
}

/* code is taken from "pafish" project, see references on the parent page */
int pafish_exists_regkey(HKEY hKey, char * regkey_s) {
    HKEY regkey;
    LONG ret;

    /* regkey_s == "HARDWARE\\ACPI\\FADT\\VBOX__"; */
    if (pafish_iswow64()) {
        ret = RegOpenKeyEx(hKey, regkey_s, 0, KEY_READ | KEY_WOW64_64KEY, &regkey);
    }
    else {
        ret = RegOpenKeyEx(hKey, regkey_s, 0, KEY_READ, &regkey);
    }

    if (ret == ERROR_SUCCESS) {
        RegCloseKey(regkey);
        return TRUE;
    }
    else
        return FALSE;
}
```

A seguir, o detalhamento de ferramentas e máquinas virtuais necessárias para o minicurso:

- **Sistema de virtualização**: Recomendamos a utilização do VMware Workstation Pro 14 (ou superior) ou VMware Fusion Pro 10 (ou superior) dependendo do sistema operacional Windows ou OS X, respectivamente. Ambos são versões comerciais. Caso o computador não possua a licença comercial do VMWare, é possível fazer o download de versões de avaliação por 30 dias. As versões Pro são necessárias uma vez que suportam a criação de snapshots - recurso essencial durante as práticas do mini-curso.

  Versões de avaliação do VMWare Pro estão disponíveis em:

  – https://www.vmware.com/products/workstation-pro.html
  – https://www.vmware.com/my/products/fusion-pro.html

  Observação importante: Computadores Apple com o processador M1 não suportam os recursos de virtualização necessários e não devem ser utilizados neste mini-curso.

- **Máquina Virtual Linux**: Para a máquina virtual Linux, recomendamos o uso da distribuição REMnux© [3]. O REMnux© é um kit de ferramentas para engenharia reversa e análise de códigos maliciosos e provê uma lista de ferramentas gratuitas criadas pela comunidade. O uso desta distribuição, portanto, facilita o trabalho do

[3] https://remnux.org/

analista uma vez que não precisa se preocupar em instalar as ferramentas de forma independente.

As instruções de instalação e o link para download do REMnux© estão disponíveis em:

- https://docs.remnux.org/install-distro/get-virtual-appliance.

- **Máquina Virtual Windows**:

  Para máquina virtual Windows, disponibilizamos um projeto chamado Win Reverse VM[4] cujo objetivo é facilitar a preparação de uma VM Windows para análise de malware.

  O projeto utiliza como base uma VM Windows 10 disponibilizada pela Microsoft por um período de avaliação. O projeto Win Reverse VM instala na VM ferramentas comumente utilizadas na análise estática e dinâmica de artefatos maliciosos. Estão na lista: Process Monitor, Process Hacker, API Monitor, x64dbg, Detect it easy, dentre outras.

  Siga as instruções de instalação do projeto para ter a sua VM Windows:

  - https://github.com/rrmarinho/winreversevm#installation

3.2.3. Configurações de isolamento

Uma vez que as VMs Linux e Windows tenham sido instaladas, certifique-se de ajustar as configurações de rede e internet de forma que:

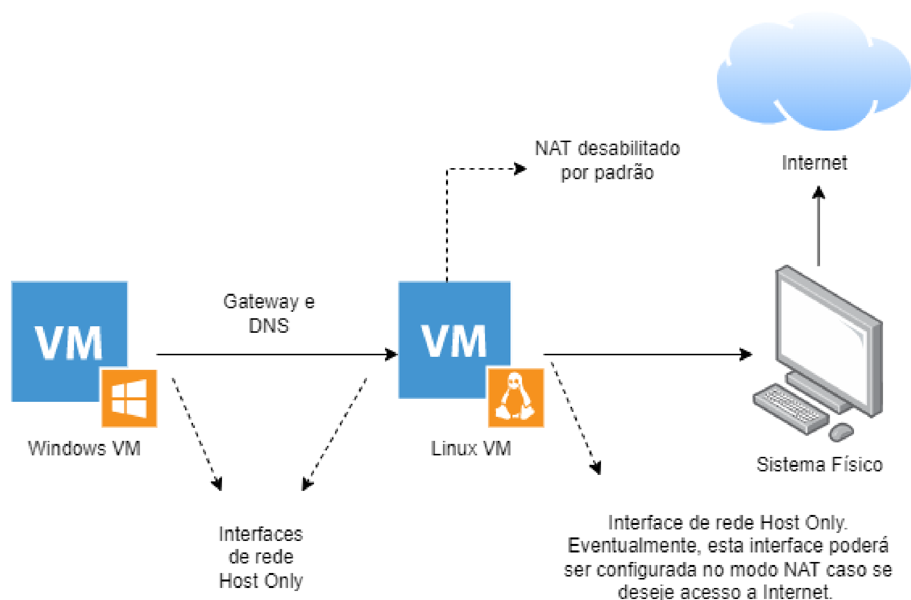
- A VM Linux tenha duas interfaces de rede - ambas em modo 'Host Only'. Uma das interfaces será utilizada para a comunicação com o host Windows e a outra para comunicação com o host hospedeiro (seu sistema operacional). A interface de comunicação com o host hospedeiro poderá ser alterada para o modo NAT caso precise de internet no host Linux ou, até mesmo, no host Windows. Para o segundo caso, será necessário o uso de NAT no Linux para habilitar a saída do Windows;
- A VM Windows tenha uma interface de rede em modo 'Host Only';
- A VM Windows tenha as configurações de Gateway e DNS apontando para o endereço IP de uma das interfaces 'Host Only' da VM Linux;
- As duas VMs não estejam saindo para a Internet.

A Figura 3.4 demonstra como deve ser montado o ambiente do laboratório.

Ao final do processo de criação das VMs Windows e Linux, crie um *snapshot* de cada das VMs para preservar o estado inicial.

[4]https://github.com/rrmarinho/winreversevm

Figura 3.4. Diagrama do Laboratório de Análise de Malware



3.3. Profiling do artefato suspeito (OSINT)

Na metodologia de análise de códigos suspeitos, a busca por informações sobre um artefato em repositórios públicos utilizando OSINT (*Open Source Intelligence*) pode fornecer desde indícios de que estejamos realmente diante de um código malicioso a análises completas, incluindo comportamento, análise reversa e indicadores de comprometimento (IOCs). Em situações de resposta a incidentes, por exemplo, onde informações tempestivas são cruciais, este atalho na análise pode fazer toda a diferença na tomada de decisões, como o isolamento de um sistema infectado ou a troca de credenciais de um usuário vítima de um *phishing*, para citar dois exemplos. Desta forma, exigindo pouco esforço e tempo do analista, a etapa de *profiling* via OSINT é, muitas vezes, a mais indicada para o início de uma análise.

Por outro lado, é importante notar que a fase de *profiling* pode não ser suficientemente rica para eliminar a necessidade de aprofundamento da análise do artefato. Alguns *samples*, como chamamos uma amostra de um *malware*, podem não ter sido analisados anteriormente ou as informações disponíveis a seu respeito podem ser muito resumidas.

Nesta seção, abordaremos o uso de técnicas de OSINT para *profiling* de artefatos suspeitos com base nos resultados do VirusTotal [5] utilizando o binário 'TextInputDocsx.exe' como exemplo. Existem muitas outras ferramentas que podem ser utilizadas, sobretudo em conjunto, neste processo, tais como Malpedia [6], Open Threat Exchange (OTX) [7], Malware Hash Registry (MHR) [8], Hybrid Analysis [9], Any.run [10] e Intezer [11].

[5] https://virustotal.com
[6] https://malpedia.caad.fkie.fraunhofer.de/
[7] https://otx.alienvault.com/
[8] https://team-cymru.com/community-services/mhr/
[9] https://www.hybrid-analysis.com/
[10] https://any.run/
[11] https://www.intezer.com/

No entanto, decidimos focar no VirusTotal por ser uma das mais abrangentes e importantes para esta finalidade. Este mesmo processo pode ser replicado para outros samples desconhecidos de seu interesse.

3.3.1. Identificação do tipo de ameaça

Uma das primeiras perguntas a serem respondidas a respeito de uma ameaça é a sua finalidade. Por exemplo, o código teria sido criado para roubar informações (*malware* do tipo *information stealer* ou simplesmente *infostealer*), seria um *backdoor* (código que oferece uma porta de entrada alternativa ao atacante) ou um ransomware? Esta pergunta está diretamente ligada ao entendimento das capacidades da ameaça e, consequentemente, às ações que devem ser tomadas em resposta a sua presença no sistema.

Nem sempre as ameaças podem ser classificadas em um único tipo. Na verdade, é comum que ameaças somem características de múltiplos tipos. Por exemplo, podem somar características de códigos que roubam informações com códigos que realizam a criptografia de disco. Desta forma, dependendo da profundidade de informações disponíveis sobre a ameaça, deveremos ter em mente que o tipo de malware apontado pode refletir somente a ação principal da ameaça. O artefato pode tomar outras ações que só podem ser descobertas com análises mais aprofundadas.

Conforme mencionado anteriormente, utilizaremos o VirusTotal na tentativa de identificação do tipo da ameaça. O VirusTotal é uma ferramenta que inspeciona o arquivo submetido com múltiplos motores de análise de malware ou *engines* (72 no momento) de anti-vírus de diferentes fabricantes. Caso o item submetido seja uma URL ou domínio de internet, a ferramenta buscará pelo item em *blocklists*. Cada *engine* analisará o artefato e, caso o identifique como malicioso, indicará um rótulo para a ameaça detectada (ex: Trojan.BitCoinMiner). Além dos rótulos, o VirusTotal poderá oferecer análises adicionais com base na execução do artefato em *sandboxes*.

O VirusTotal fornece três opções de uso da ferramenta no seu *website*:

1. FILE: utilizando a opção 'FILE', o usuário da ferramenta submete o arquivo para análise pelas múltiplas *engines*. É muito importante que o usuário esteja atento ao aviso do VirusTotal de que, ao fazer isso, estará compartilhando o arquivo com os assinantes da plataforma VirusTotal. Isso significa que o usuário poderá correr o risco de ter dados sigilosos expostos a terceiros. Por isso, esta não seria a opção inicial mais adequada;

2. URL: esta opção permite que o usuário forneça uma URL que será avaliada por múltiplas *engines* e *blocklists*. Esta consulta também será compartilhada com os assinantes da plataforma;

3. SEARCH: Nesta opção, o usuário tem a possibilidade de fazer uma busca na plataforma por análises prévias de URLs/domínios, endereços IP e *hashes* de arquivos. Observe que, ao invés de submeter o arquivo em si, podemos submeter o identificador *hash* do arquivo. Caso o arquivo já tenha sido submetido por outro usuário, você terá acesso ao resultado da análise.

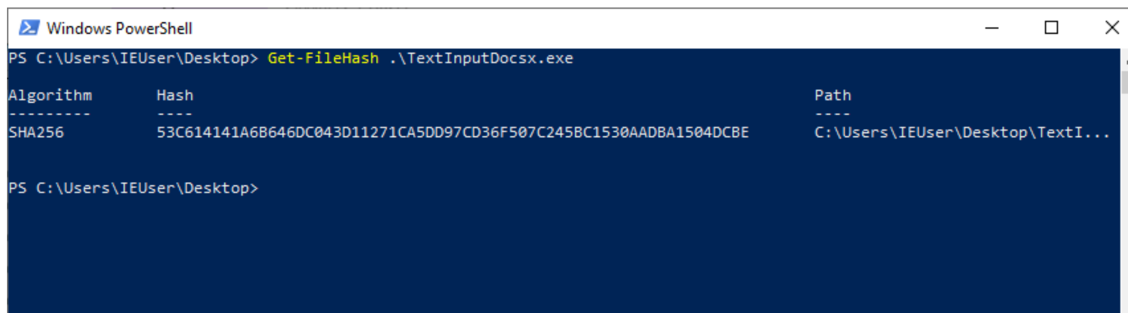
© 2022 Sociedade Brasileira de Computação

Como pode ser observado, a opção SEARCH é a mais adequada para o início da etapa de *profiling* do artefato utilizando o VirusTotal.

Para ilustrar como o VirusTotal pode auxiliar na identificação do tipo da ameaça, vamos submeter à ferramenta o *hash* MD5 do binário ’TextInputDocsx.exe’ à plataforma. Para calcular o *hash* do binário, há diferentes opções. Na VM Windows, pode utilizar o utilitário DIE ou simplesmente um PowerShell *Get-FileHash*. A seguir, como o comando PowerShell pode ser utilizado:

1. Abrir um prompt de comando PowerShell;
2. Mudar o diretório para o local onde o binário ’TextInputDocsx.exe’ encontra-se no disco;
3. Executar o comando: GetFileHash .\TextInputDocsx.exe. Por padrão, o algoritmo *hash* SHA256 será utilizado. O SHA256 pode ser utilizado na busca do VirusTotal. No entanto, caso queira calcular com o algoritmo MD5, basta passar o parâmetro -a MD5. Assim, teríamos: GetFileHash -a MD5 .\TextInputDocsx.exe. Na Figura 3.5, temos o exemplo de uso do comando.

Figura 3.5. Cálculo do hash do binário TextInputDocsx.exe usando o comando Get-FileHash



O *hash* SHA256 resultante é o: 53C614141A6B646DC043D11271CA5DD97CD36F507C245BC1530AADBA1504DCBE. Utilizando este código na busca do VirusTotal, tivemos o resultado apresentado na Figura 3.6.

A partir desta resultado, podemos observar:

- O VirusTotal nos informa que 50 de 69 *engines* consideraram o arquivo malicioso. Quando um número muito baixo de *engines* consideram o arquivo malicioso, há uma chance de entendermos que o arquivo seja malicioso mesmo que ele não seja. No entanto, quando temos um grande número de *engines* apontando que o arquivo é malicioso, como neste caso, a chance de cairmos num falso positivo é bem menor;
- A análise mais recente do VirusTotal para este arquivo foi no dia 03/10/2019 às 18:11:20 UTC. Este indicador pode nos dizer que esta ameaça não é algo novo,

© 2022 Sociedade Brasileira de Computação

Figura 3.6. Resultados do VirusTotal para o hash do arquivo TextInputDocsx.exe

ou, pelo menos, que não há uma campanha abrangente em curso utilizando esta ameaça. Para ameaças utilizadas em campanhas abrangentes ou genéricas, é mais comum termos uma análise recente da ameaça no VirusTotal;

- Analisando os rótulos associados pelas *engines* de antivírus, encontramos múltiplos deles indicando que o binário trata-se de um Trojan. Adicionalmente, 5 dos rótulos indicam que o artefato seria um *Banker*, ou seja, um malware bancário cuja finalidade principal é o roubo de dados financeiros das vítimas. Por fim, 8 das engines indicam no rótulo a palavra *Banbra*[12]. Este rótulo indica que este é um malware projetado para roubar informações pessoais de clientes de bancos brasileiros.

3.3.2. Identificação de capacidades da ameaça

Em suma, até aqui, com a simples consulta do *hash* do binário no VirusTotal, pudemos entender, a partir de um repositório compartilhado de dados, que o binário suspeito trata-se de um **trojan bancário com alvo em clientes de bancos brasileiros**.

Mas há ainda mais possibilidades de descobertas a partir do VirusTotal. Uma delas é a possível descoberta de indicadores de comprometimento (IOCs). IOC, em cibersegurança, é um artefato observado na rede ou em um sistema que, com alta confiança, indica o comprometimento por uma ameaça em particular [Gragido 2012]. Caso estejam disponíveis, estarão principalmente nas abas 'BEHAVIOR' e 'RELATIONS'.

Em 'BEHAVIOR', teremos acesso aos resultados da execução do binário em uma ou mais *sandboxes*. Na Figura 3.7, podemos observar a aba 'BEHAVIOR' para o nosso artefato 'TextInputDocsx.exe' na qual há indicadores de comunicações de rede: uma conexão HTTP e 2 conexões DNS. Voltando ao tema de identificar IOCs importantes para

[12]https://threats.kaspersky.com/br/threat/Trojan-Banker.Win32.Banbra/

o nosso artefato, certamente estes são de grande valia. Alguém que queira buscar por computadores que tenham executado a ameaça, pode buscar por computadores que tenham tentado resolver os nomes freedow.ml e livekernelreports.duckdns.org a partir dos registros de log do DNS Server e pode buscar por computadores que tenham enviado a comunicação HTTP para a URL http://livekernelreports.duckdns.org/ups/.

Figura 3.7. Aba BEHAVIOR do VirusTotal para o binário TextInputDocsx.exe

Na aba ’RELATIONS’, você poderá ter acesso a análise dos IOCs desta ameaça pelo VirusTotal de forma individualizada. Para o nosso exemplo, teremos acesso nesta aba às detecções dos domínios resolvidos nas consultas DNS e da URL acessada via HTTP. Adicionalmente, teremos acesso aos endereços IP para os quais as consultas DNS foram resolvidas no momento da execução do malware na sandbox. A informação dos endereços IP agregam aos IOCs de domínio e URL já observados.

Por fim o VirusTotal oferece dados que podem ser igualmente relevantes para a nosso objetivo de *profiling*. Na aba ’COMMUNITY’, o VirusTotal apresenta possíveis comentários e referências externas citando a ameaça. Para ter acesso a informações mais completas desta aba, crie uma conta gratuita no VirusTotal. Para o exemplo do binário ’TextInputDocsx.exe’, não há referências externas contendo análises. No entanto, para ilustrar as possibilidades, vamos considerar a análise de um outro binário cujo *hash* MD5 é e74ec7381bf7020ec707bd722afe6d38. A Figura 3.8 mostra a aba ’COMMUNITY’ com links para uma análise da ameaça realizada por Renato Marinho cujo título é *Example of how attackers are trying to push crypto miners via Log4Shell* [13]. Ao acessar referências externas, poderá ter ainda mais indicadores ou mesmo um entendimento mais abrangente das capacidades da ameaça, como neste caso.

[13]https://morphuslabs.com/example-of-how-attackers-are-trying-to-push-crypto-miners-via-log4shell-294ec2a5cf43

Figura 3.8. Exemplo de referências externas para uma ameaça no VirusTotal

3.4. Análise automatizada

Neste seção, serão abordados conceitos e ferramentas para a realização de análise automatizada de artefatos suspeitos. Ferramentas de análise automatizada podem ser bastante úteis uma vez que, sem muito esforço por parte do analista, informações valiosas de comportamentos maliciosos, como, por exemplo, a criação de uma chave de registro que fará com que o ameaça se torne persistente no sistema operacional (autoexecutável no *reboot*), possam ser obtidas em um curto espaço de tempo. Entretanto, diferentes plataformas apresentam especificidades quanto ao alvo da execução em sandbox. Como exemplo, enquanto na plataforma Windows exemplares de malware afetam o subsistema de chaves de registro [Botacin et al. 2018], na plataforma Linux, exemplares de malware afetam o subsistema de arquivo /proc [Galante et al. 2018].

No entanto, os ambientes automatizados também apresentam limitações. Algumas ameaças implementam técnicas *anti-sandbox* e não manifestam suas ações maliciosas ao perceberem que estão em um ambiente de análise. Adicionalmente, o ambiente de *sandbox* pode não apresentar todos os recursos ou estímulos necessários para que o artefato malicioso manifeste seus comportamentos. Por exemplo, a ameaça pode fazer um teste da linguagem do teclado do usuário. Caso não esteja em ptBR, não executará o estágio seguinte. Outros comportamentos mais avançados podem exigir a comunicação com um servidor remoto de comando e controle (C2), o que poderá não estar disponível em um ambiente de *sandbox*. Apesar destas limitações, dada a boa relação esforço/benefício, ou seja, baixo esforço para o potencial benefício, esta etapa pode ser considerada nas etapas iniciais da análise de um artefato suspeito.

3.4.1. Funcionamento da análise automatizada

A análise automatizada consiste na execução de arquivos suspeitos em ambientes controlados (*sandboxes*) com o objetivo de identificar comportamentos maliciosos. Enquanto o arquivo suspeito é executado, o ambiente da *sandbox* monitora cada evento gerado pelo artefato no sistema e busca identificar comportamentos potencialmente maliciosos, como a mudança de um registro do Windows que faz com que algo seja executado na iniciali-

© 2022 Sociedade Brasileira de Computação

zação do sistema (técnica de persistência).

Finalizada a execução do artefato, o sistema que provê a *sandbox* emitirá um relatório das ações observadas com destaque especial para aquelas potencialmente maliciosas. Adicionalmente, os relatórios apresentam uma lista de indicadores de comprometimento (IOCs), tais como endereços IP ou nomes de *hosts* consultados durante a execução, arquivos ou registros criados/acessados no sistema, etc. No geral, os relatórios trazem também um percentual indicando um grau de 'certeza' de que aquele artefato seria malicioso com base nos seus comportamentos.

Caso o resultado indique que o artefato é malicioso e aponte suas características, o analista já terá informações valiosas para a tomada de decisões sobre os próximos passos na estratégia de defesa. Caso o resultado aponte que o artefato não é malicioso, é importante ter em mente que este resultado pode não refletir a realidade. Alguns artefatos maliciosos podem identificar que estão em um ambiente controlado e podem não manifestar suas ações ou podem reagir com ações que parecem legítimas para despistar os resultados da *sandbox*, conhecidas como técnicas *anti-sandbox*. Neste caso, o mais indicado é que o analista busque avançar em análises mais aprofundadas para descobrir se o artefato é malicioso ou não.

Há esforços da comunidade acadêmica para minimizar as chances de detecção do ambiente controlado. Em [Liu et al. 2022], os autores propõem a emulação de comportamentos de usuários reais no sistema. Já a pesquisa [Mills and Legg 2020] investiga as técnicas *anti-evasion* por meio de reconfiguração da *sandbox*.

3.4.2. Tipos de sandboxes

Há uma grande variedade de ferramentas para análise automatizada de malware com o uso de *sandboxes*. Essas ferramentas podem ser divididas em dois grupos principais: *sandboxes* públicas e *sandboxes* privadas.

As *sandboxes* públicas são aquelas fornecidas por provedores de serviços em cibersegurança que disponibilizam suas *sandboxes* para usuários via internet. A maior parte destas plataformas aplicam alguma limitação na versão gratuita da *sandbox*, como, por exemplo, o tempo que o binário suspeito passará em execução.

As *sandboxes* públicas apresentam vantagens como a facilidade de uso e a disponibilidade imediata. Como desvantagens, podemos citar as limitações para as versões gratuitas e, principalmente, a necessidade da submissão do *sample* a ser analisado. Nestes casos, é importante observar se o arquivo terá que ficar disponível para terceiros.

A seguir, uma lista das principais ferramentas de *sandbox* disponíveis online:

- Any.run (https://any.run)
- Intezer Analyze (https://analyze.intezer.com/)
- Hybrid Analysis (https://www.hybrid-analysis.com/)
- Joe Sandbox Cloud (https://www.joesandbox.com/)
- FileScan.IO (https://www.filescan.io/)

Do outro lado estão as *sandboxes* privadas. Existem as soluções corporativas e soluções *open-source*. As soluções corporativas, geralmente fornecidas por provedores de soluções de anti-malware, tem a vantagem da facilidade de implementação e do suporte ao usuário, mas pode ser mais limitada quanto a customizações. Já a solução open-source, pode apresentar desafios de implantação e pode ser mais limitada em recursos, mas tem a grande vantagem da customização.

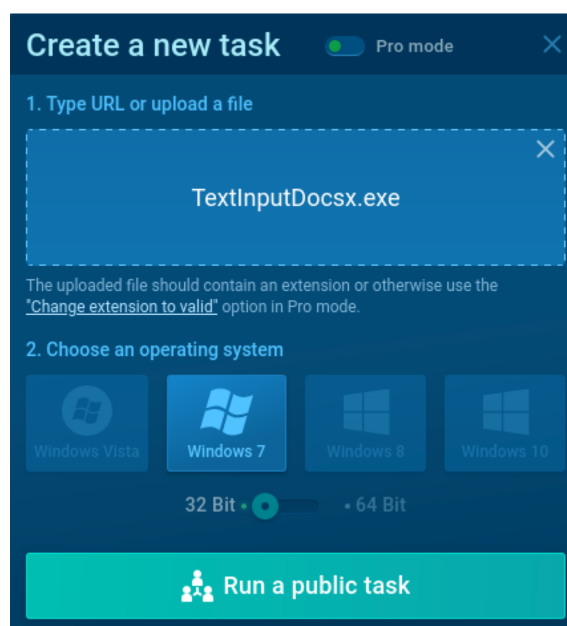
Uma ferramenta de *sandbox* open-source bastante popular é a Cuckoo Sandbox[14]. A ferramenta pode analisar diferentes tipos de arquivos (executáveis, documentos office, arquivos PDF, e-mails, etc), bem como websites maliciosos nas plataformas Windows, Linux, macOS e Android por meio de ambientes virtualizados.

3.4.3. Analisando nosso artefato utilizando sandboxes

Utilizaremos a *sandbox* Any.run para analisar o nosso binário de exemplo (TextInput-Docsx.exe). Para isso, tivemos que criar uma conta gratuita na plataforma para, então submeter o arquivo. Seguimos estes passos utilizando a VM Linux, uma vez que é mais fácil ter acesso à internet por meio desta plataforma. Lembramos que não é recomendável descompactar o binário malicioso no seu sistema real por dois principais motivos: primeiro que o seu antivírus poderá reconhecê-lo como malicioso e apagá-lo. Segundo porque você pode clicar no binário por acidente descompactado e executar o malware no seu host.

Ao submetermos o binário ao Any.run, verificamos as limitações da versão gratuita: execução somente no Windows 7 e na arquitetura 32bits e a disponibilização dos dados para acesso público, conforme Figuras 3.9 e 3.10.

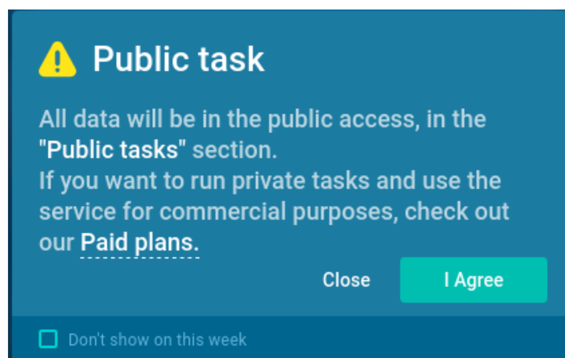
Figura 3.9. Limitações de execução da versão grátis da Sandbox Any.run



Como resultados para a análise, podemos destacar:

[14]https://cuckoosandbox.org/

Figura 3.10. Disponibilização dos dados submetidos para acesso público



- As consultas DNS para os domínios livekernelreports.duckdns.org e freedow.ml;
- Tentativas de conexão ao endereço IP 188.114.97.3 (resolvido para a consulta do domínio freedow.ml) na porta TCP/26457;
- Aviso de ameaça em potencial devido a tentativa de resolução de um domínio suspeito (.ml) e um domínio hospedado em um serviço de DNS dinâmico (*.duckdns);
- Há a mudança/criação do arquivo 'C:\Users\admin\AppData\Roaming\Progrestime.html';
- O artefato faz a leitura de chaves de registro para verificar os idiomas suportados bem como para obter o nome do computador;

Como resultado final, apesar de alguns indicadores potencialmente maliciosos, não há um veredito da plataforma. A análise do *sample* 'TextInputDocsx.exe' atingiu uma pontuação de 30 de um total de 100, o que não foi suficiente para apontar que o binário é malicioso.

Analisamos também o artefato utilizando a *sandbox* Hybrid Analysis e nesta o resultado apontou que o binário é malicioso. A pontuação desta *sandbox*, chamada de *Threat Score* atingiu 100 de 100 possíveis. Como destaques para o relatório gerado, temos:

- O *sample* foi considerado malicioso por uma grande quantidade de *engines* de antivírus;
- O processo submete arquivos a um *web server*;
- O processo aloca memória em processos remotos;
- Utiliza URLs identificadas como maliciosas por pelo menos uma *engine* de reputação;
- Por fim e não menos importante, que o processo tem a habilidade de capturar as teclas digitadas no teclado. Esta é uma característica comum de códigos maliciosos que visam o roubo de informações.

Diferentemente da Any.run, a Hybrid Analysis chegou a veredito de que o artefato é malicioso e apresentou pelo menos um IOC a mais, que foi a tentativa de conexão HTTP a URL 'livekernelreports.duckdns.org/ups/'.

Com isso, observamos que a análise do nosso binário com o uso destas duas *sandboxes* gratuitas agregou algumas novas informações e indicadores ao que já havíamos conseguido de informações na fase de *profiling*. Sabemos agora que o processo busca a alocação de memória em outros processos numa possível tentativa de persistência e evasão de defesas e que tem a habilidade de capturar teclas digitadas pelo usuário.

3.5. Análise estática

A análise estática é fundamental para uma triagem bem sucedida de um artefato desconhecido. A partir dela é possível identificar indicadores estáticos e propriedades suspeitas do artefato e fornecer um direcionamento para as próximas fases de análise: a comportamental e a reversa. Essa fase pode ter etapas prejudicadas por ofuscações existentes no artefato, tais como o uso de *packers*, que são softwares que ofuscam o conteúdo através do uso de criptografia ou compressão.

Essa fase é essencialmente importante em cenários de incidentes no qual o sample a ser analisado não consta em bases já conhecidas de malwares, podendo ser uma ameaça completamente nova ou apenas a modificação ou variante de uma já existente. Nas seções que seguem, será explicitada cada etapa de análise estática a ser realizada em um artefato.

3.5.1. Identificação do tipo de arquivo

Uma das primeiras análises estáticas que deve ser feita em um artefato desconhecido, é identificar seu tipo de arquivo. Todos os arquivos são compostos de zeros e uns e sua diferenciação está no seu formato, ou padrão de construção. Um programa que aceita um tipo de arquivo, como um editor de imagens ou reprodutor de áudio, realiza um processo de *parsing* no arquivo para ler seu conteúdo.

Arquivos executáveis são lidos e interpretados por um programa que reside no sistema operacional, chamado de *loader*. O loader irá preparar os requisitos de endereçamento virtual, memória, registradores e dependências para a criação de um processo, ou *imagem*, do executável.

Arquivos *EXE* e *DLL*, por exemplo, utilizam o formato **Portable Executable (PE)**, enquanto que sistemas Linux e BSD utilizam **Executable and Linking Format (ELF)** e o sistema macOS faz uso do **Mach object file format (Mach-O)**.

Na Figura 3.11, está exemplificado as estruturas que compõem um binario PE. Nela, é possível perceber que

Abaixo, na Figura 3.12, podemos identificar a composição dos bytes de um arquivo PE em disco. Algumas estruturas importantes são o *DOS Header*, que funciona como uma camada de compatibilidade com o *MS-DOS*, o DOS Stub, que é um pequeno programa em MS-DOS executado apenas quando o programa está rodando no MS-DOS e o PE Header, que conterá metadados necessários para instruir o loader a como carregar o binário.

Figura 3.11. Estrutura geral de um *Portable Executable*

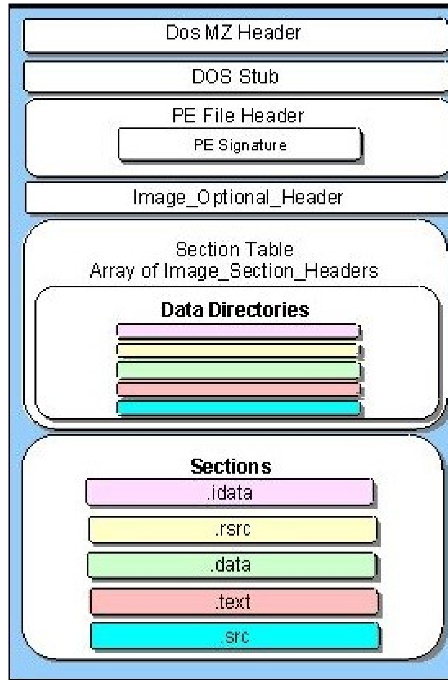


Figura 3.12. Estrutura geral de um *Portable Executable*

DOS HEADER
PE OFFSET
DOS STUBB
PE HEADER

PE HEADER

IMAGE_NT_HEADER (PE SIGNATURE): 4 BYTES (50 45 00 00)

IMAGE_FILE_HEADER: 20 BYTES

IMAGE_OPTIONAL_HEADERS: 240 BYTES

Magic: 0x10B0 (32 bits) 0x20B0 (64 bits)

AddressOfEntryPoint: address where windows loader will begin execution.

Todo arquivo PE começa com os bytes "MZ", e nos quatro bytes (também chamados de *DWORD*) começando no *offset* 0x3C, há o campo *e_lfanew* do DOS Header. Esse

valor, armazenado em *little endian*, é a localização em disco para a *PE Signature*, que são os bytes "PE\0\0", e identificam aquele arquivo como um *Portable Executable*. A PE Signature está contida no *IMAGE_NT _HEADERS*, um cabeçalho que abrange todos os demais cabeçalhos referentes ao formato.

Imediatamente após a assinatura PE, há o início do cabeçalho *IMAGE_FILE _HEADERS*[15], que contém 20 bytes de tamanho, responsável por armazenar informações da arquitetura alvo, o número de seções, o timestamp de compilação, além de informações que auxiliam a construção dos demais headers.

```
typedef struct _IMAGE_FILE_HEADER {
  WORD  Machine;
  WORD  NumberOfSections;
  DWORD TimeDateStamp;
  DWORD PointerToSymbolTable;
  DWORD NumberOfSymbols;
  WORD  SizeOfOptionalHeader;
  WORD  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Após o *IMAGE_FILE_HEADERS*, há o *IMAGE_OPTIONAL_HEADER*[16], de tamanho variável, definido pelo campo *SizeOfOptionalHeader* do cabeçalho anterior. Os campos mais úteis uma análise de malware estão na listagem abaixo.

```
typedef struct _IMAGE_OPTIONAL_HEADER {
...
  DWORD                AddressOfEntryPoint;
...
  DWORD                ImageBase;
...
  WORD                 Subsystem;
  WORD                 DllCharacteristics;
...
  IMAGE_DATA_DIRECTORY DataDirectory
  [IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

O campo *AddressOfEntryPoint* conterá o *offset*, que será somado ao valor do *ImageBase* o, e essa soma irá resultar no endereço virtual do primeiro byte a ser executado durante o carregamento do EXE. Em DLLs, não é necessário um endereço de entrada, visto que elas são carregadas de maneira diferente que EXEs.

Os campos *Subsystem* e *DllCharacteristics* definem, respectivamente, qual o subsistema de execução, por exemplo, se a aplicação roda com gráficos ou apenas modo

[15]https://docs.microsoft.com/en-us/windows/win32/api/winnt/nswinntimage_file_header#syntax
[16]https://docs.microsoft.com/en-us/windows/win32/api/winnt/nswinntimage_optional_header32#syntax

texto, e informações sobre proteções de carregamento de códigos em endereços de dados (*Data Prevention Execution - DEP*) e randomização de endereços de carregamento (*Address Space Layout Randomization - ASLR*). Por fim, o campo *DataDirectory* definirá um array fixo de diretórios de dados, que são tabelas que contém metadados de importação, exportação, relocação, entre outros. É nessa estrutura que serão descritas os endereços e tamanhos de cada um dos diretórios de dados.

As seções são regiões de bytes bem definidos que têm um propósito específico quando carregadas, como execução de códigos e armazenamento de dados inicializados, não inicializados e somente leitura, para citar algumas. Os nomes e funcionamentos das seções podem variar de compilador para compilador, mas cada uma delas terá informações de tamanho em disco e virtual e permissões para as páginas de memória (leitura, escrita e execução). Na listagem abaixo, segue o nome das principais seções presentes na maioria dos binários PE e uma descrição sobre sua funcionalidade.

- **.text**: contém instruções codificadas para uma determinada arquitetura, geralmente com permissões de leitura e execução;
- **.rdata**: armazena dados inicializados que são somente leitura;
- **.data**: armazena dados inicializados, como variáveis globais, que permitem leitura e sobrescrita;
- **.idata**: quando presente, conterá informações de funções importadas;
- **.edata**: guarda informações sobre funções exportadas pelo binário;
- **.pdata**: apenas presente em binários 64-bits, guarda informações sobre tratamento de exceções;
- **.rsrc**: armazena informações sobre imagens, strings, ícones e outros recursos embutidos em um binário.

3.5.2. Packers

Um *packer* é um software que irá modificar o conteúdo de um executável, comprimindo ou ofuscando seções de código e de dados. Ao realizar esse procedimento, o packer também colocará um *stub*, código a ser executado para a descompressão ou desofuscação do código original em runtime, e irá alterar o entry point do binário para o endereço desse procedimento. Quando o binário é executado, há a desofuscação do código original através da execução do stub, e após esse procedimento, o stub irá alterar o fluxo de código para o código original.

Packers conhecidos, como o *Ultimate Packer for eXecutables (UPX)*[17], contém assinaturas catalogadas por soluções antivírus e identificadores de tipos de arquivos. Através da detecção do packer, é possível aplicar algumas técnicas para retirá-lo do binário, que vão desde buscar por softwares *unpacker*, que realizam a desofuscação do código original

[17]https://github.com/upx/upx

© 2022 Sociedade Brasileira de Computação

e retiram o *stub*, até o *unpacking* manual do código, buscando executar e encontrar o fim do *stub* e o entry point original, através de análise reversa.

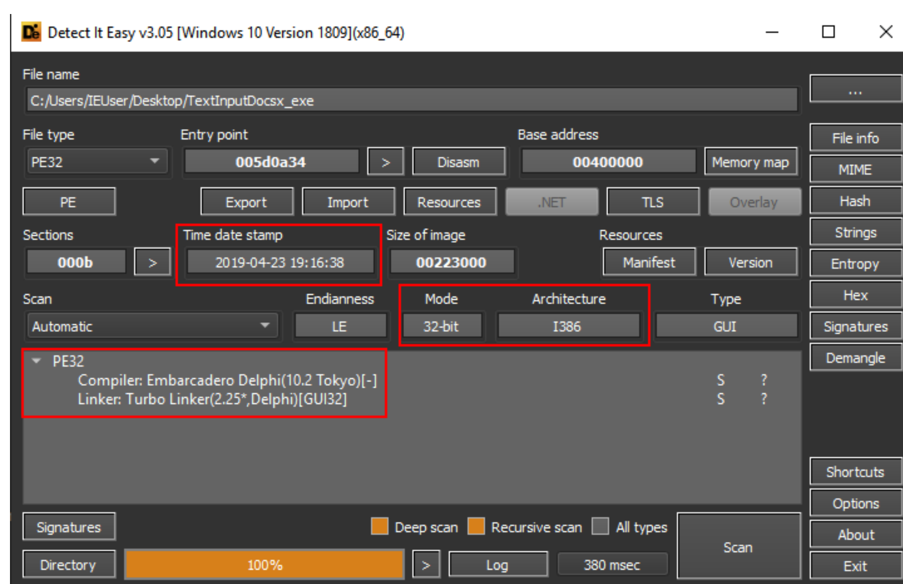
Além de assinaturas, a entropia, que é a medida de desorganização de um determinado sistema, também pode ser útil em cenários de análise de malware. Ela pode servir para identificar códigos criptografados ou comprimidos, visto que os bytes que compõem as instruções de um código estão dispostos de uma forma mais organizada, e com diversos padrões repetidos, do que um código ofuscado de alguma forma. Isso permite inferir que determinado binário utiliza um packer, mesmo quando esse não tem assinatura catalogada. Não há um valor de entropia que identifique que um binário utiliza um packer com certeza, logo, o analista deverá levar em consideração outros fatores que corroborem para essa hipótese, como o baixo nível de strings legíveis por humanos e poucas funções utilizadas.

3.5.3. Prática de identificação e parsing de arquivo PE

Para a tarefa de identificação e *parsing* de um artefato, é possível utilizar o detector de tipos e parser PE *Detect It Easy (DIE)*. Na ferramenta, é possível identificar o tipo de arquivo, navegar por sua estrutura PE (caso seja um binário PE), buscar informações de seções de código e de dados, strings, entropia e diversos outros conceitos que serão abordados nas próximas seções.

Na identificação do artefato 'TextInputDocsx.exe' a partir da ferramenta DIE, é possível verificar o modo de operação (32-bit ou 64-bit), a arquitetura, identificação do compilador (Delphi, nesse caso) e a data e o horário em que possivelmente o binário foi compilado, conforme Figura 3.13.

Figura 3.13. Informações gerais do binário ao ser aberto no identificador DIE

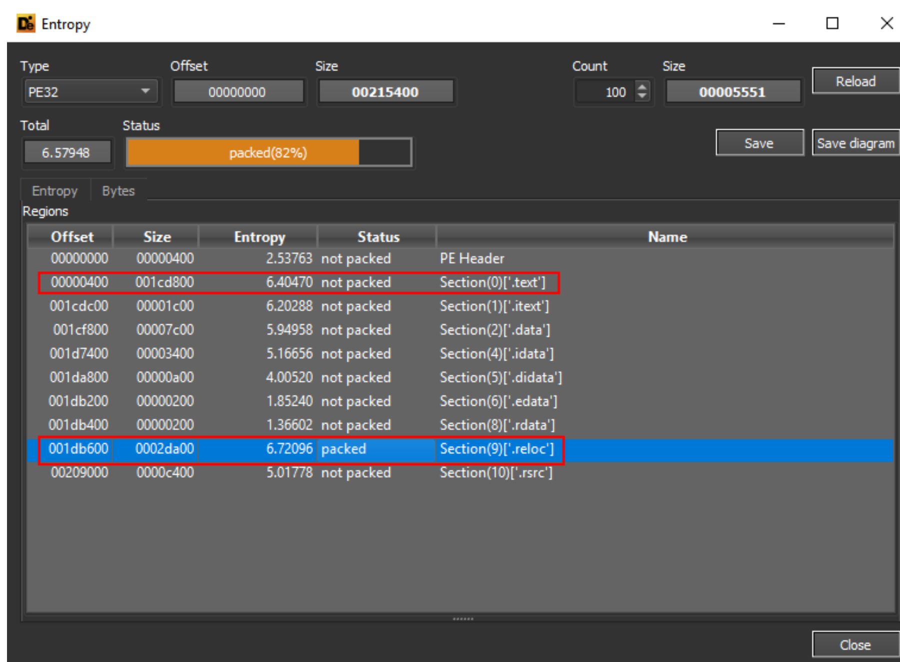


Foi citado data e horário que possivelmente o binário foi compilado pois na compilação de um código-fonte, o compilador irá produzir um binário correspondente e escreverá um tempo de compilação nos metadados do arquivo. Porém esse valor pode ter sido alterado por um ator malicioso para falsificar a data original. Além disso, certos

compiladores utilizam datas padrão para todos os binários, que não reflete o período real de compilação.

Ao verificar a entropia, é possível identificar que o binário contém seções com baixa entropia e uma, a *.reloc*, que está acima da média para uma seção comum. Devido a isso, a ferramenta informa que o binário, como um todo, está possivelmente com *packer*. Entretanto, como apenas uma seção está com uma alta entropia, e ela não é a de código (*.text*) ou de dados (*.data*), podemos inferir que o binário não está usando um packer.

Figura 3.14. Entropia das seções do binário



3.5.4. Identificação de strings maliciosas

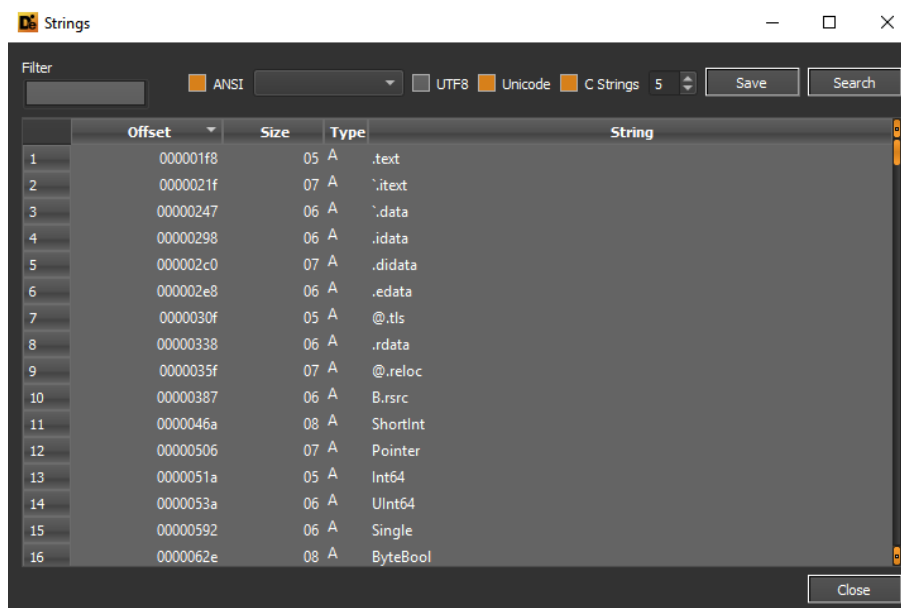
As ferramentas *DIE* e *PEStudio* permitem a busca de cadeias de caracteres incluídas em arquivos binários. Dentro de um binário podem ser encontradas URLs, IPs, domínios, chaves de registro, entre outros possíveis indicadores maliciosos. Também é possível identificar certos tipos de encoding e criptografia através de certas cadeias de caracteres padrão, como no caso da codificação *base64*, que pode ser identificada por uma cadeia de caracteres que contém os intervalos de letras, símbolos e números A–Z, a–z, 0–9, +, / e =.

Além disso, é importante notar a distinção entre cadeias de caracteres *ASCII* e *UTF-16*. Strings ASCII, mais comuns em programas Linux, contém apenas um byte por caractere e terminam com o valor \0, também chamado de *NULL*. Strings *UTF-16*, por sua vez, são prevalentes na programação para a plataforma Windows, contém dois bytes por caractere, sendo o segundo byte o valor \0, e terminam pelo caractere \0\0. Dependendo da ferramenta utilizada para busca das strings, será necessário indicar a codificação na qual deseja-se buscar as strings.

Na Figura 3.15 está uma imagem da ferramenta *DIE*, na janela de busca de strings do artefato 'TextInputDocsx.exe'. É possível buscar por strings através do botão "Strings",

no qual uma janela irá se abrir, mostrando strings ASCII e UTF-16 por padrão. Nesse artefato em específico, não foi possível encontrar strings que fossem interessantes para *insights* sobre o funcionamento dele.

Figura 3.15. Strings encontradas através do DIE



3.5.5. Imports e funções

Todo binário *PE* tem uma estrutura chamada *import table*, localizada nos *Data Directories* e representada pela seção *.idata*. Nela vão estar informações de quais funções externas contidas em DLLs o loader deverá carregar para o funcionamento correto do executável.

Todo EXE deverá carregar a *kernel32.dll* e algumas outras DLLs, que dependerão das funcionalidades escritas pelo programador. Logo, a partir da import table, é possível inferir quais são as funcionalidades de um artefato. Por exemplo, caso o binário crie subprocessos, na sua import table provavelmente constará a função *ShellExecute* da *Shell32.dll*, ou caso faça conexão com um host externo, o uso da função *gethostbyname* da *Winsock2.dll* poderá indicar a resolução *Domain Name System (DNS)* de um dado nome de host. Esse processo de importar funções externas ao binário e consequentemente armazenar o nome das bibliotecas e funções importadas na import table é chamado de *dynamic linking*.

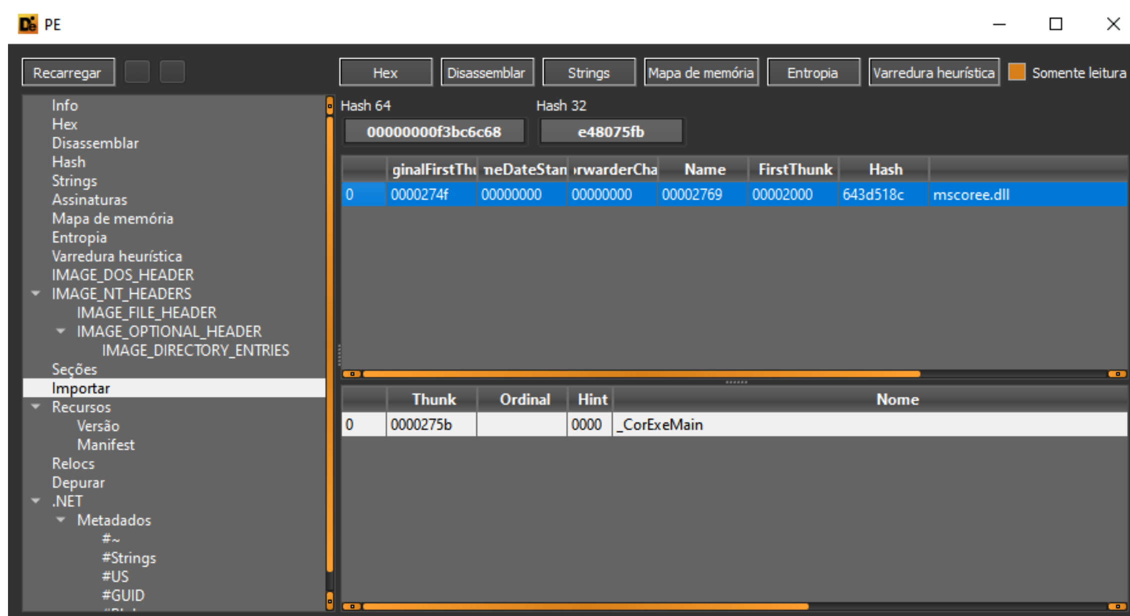
Há algumas abordagens para evitar o uso *dynamic linking*. Uma delas é o *static linking*, onde o código das funções externas utilizadas são copiadas para o programa, durante o processo de geração do binário, não gerando nenhuma dependência externa de bibliotecas e por esse motivo não tendo tais funções descritas na import table. Outra forma é o *runtime linking*, onde serão utilizadas as funções *GetProcAddress* e *LoadLibrary* da *kernel32.dll* para resolução dos endereços das funções durante a execução do programa, evitando guardar seus nomes na importa table.

Em um binário simples, como uma aplicação que apenas escreve na tela *Hello World!*, há uma série de imports implícitos, feitos pelo compilador no processo de ge-

ração do executável. Tais imports são necessários para o carregamento correto de até mesmo binários simples, e uma import table que contenha poucos imports, especialmente as que contém *GetProcAddress* e *LoadLibrary*, usadas no *runtime linking*, indicam que determinado binário provavelmente está ofuscado de alguma forma.

Algumas linguagens de programação implementam parte da funcionalidade de *runtime* dos binários em DLL, e a presença de tais DLLs ou funções pode indicar o uso de determinada linguagem na construção do binário. Por exemplo, é possível identificar o uso da linguagem C# em um binário pela presença da DLL *mscoree.dll*, com a função importada *_CorExeMain*, como exemplificado na Figura diefig:dotnet_die, através de um simples programa que escreve 'Hello World!' na tela.

Figura 3.16. Exemplo da import table de um binário escrito em C#



3.5.6. Capacidades do binário com CAPA

A ferramenta CAPA, desenvolvida e distribuída pela Fireeye, permite a análise de capacidades suspeitas ou maliciosas de EXEs ou DLLs. Ela funciona a partir de uma base de arquivos de assinatura, escritos no formato *YAML Ain't Markup Language (YAML)*, que buscam por padrões já identificados e catalogados de comportamentos suspeitos ou maliciosos. Com ela, é possível identificar se um binário realiza *dynamic linking*, tem funções de keylogging ou conecta-se com um servidor de *command and control (C2)*, entre outras capacidades.

Ao executar *C:\Tools\Reverse\capa\capa.exe TextInputDocsx.exe*, há o processamento do binário e suas funções pelo capa e são retornadas três tabelas, que constroem diferentes visões das capacidades do artefato. A primeira tabela, conforme Figura 3.17, mostra uma visão do artefato dividida em táticas e técnicas do framework *MITRE ATT&CK*. É possível notar que a ferramenta identificou capacidades de coleta de entradas do usuário, da tela, além de capacidades relacionadas a evasão de virtualização/sandbox e descoberta

de arquivos da máquina.

Figura 3.17. Correlações *MITRE ATT&CK* retornadas pela ferramenta *capa*

```
+------------------------+------------------------------------------------------------------------------------+
| ATT&CK Tactic          | ATT&CK Technique                                                                   |
|------------------------+------------------------------------------------------------------------------------|
| COLLECTION             | Input Capture::Keylogging T1056.001                                                |
|                        | Screen Capture:: T1113                                                             |
| DEFENSE EVASION        | File and Directory Permissions Modification:: T1222                                |
|                        | Hide Artifacts::Hidden Window T1564.003                                            |
|                        | Obfuscated Files or Information:: T1027                                            |
|                        | Obfuscated Files or Information::Indicator Removal from Tools T1027.005            |
|                        | Virtualization/Sandbox Evasion::User Activity Based Checks T1497.002               |
| DISCOVERY              | Application Window Discovery:: T1010                                               |
|                        | File and Directory Discovery:: T1083                                               |
|                        | Query Registry:: T1012                                                             |
|                        | System Information Discovery:: T1082                                              |
|                        | System Location Discovery:: T1614                                                  |
|                        | System Location Discovery::System Language Discovery T1614.001                     |
| EXECUTION              | Command and Scripting Interpreter:: T1059                                          |
|                        | Shared Modules:: T1129                                                             |
+------------------------+------------------------------------------------------------------------------------+
```

A segunda tabela traz informações sobre *Malware Behavior Catalog (MBC)*, catalógo de comportamentos de malwares baseado no *MITRE ATT&CK*. No caso do artefato, é possível verificar a utilização de técnicas para detecção de debuggers via API *GetTickCount*, de virtualização, criação de *mutexes*, e corroborar com a capacidade de *keylogging/screenlogging*.

Figura 3.18. Correlações *MBC* retornadas pela ferramenta *capa*

```
+-----------------------------+---------------------------------------------------------------------------------+
| MBC Objective               | MBC Behavior                                                                    |
|-----------------------------+---------------------------------------------------------------------------------|
| ANTI-BEHAVIORAL ANALYSIS    | Debugger Detection::Timing/Delay Check GetTickCount [B0001.032]                 |
|                             | Virtual Machine Detection::Human User Check [B0009.012]                         |
| ANTI-STATIC ANALYSIS        | Disassembler Evasion::Argument Obfuscation [B0012.001]                          |
| COLLECTION                  | Keylogging::Polling [F0002.002]                                                 |
|                             | Screen Capture::WinAPI [E1113.m01]                                              |
| CRYPTOGRAPHY                | Encrypt Data::RC4 [C0027.009]                                                   |
|                             | Generate Pseudo-random Sequence:: [C0021]                                       |
|                             | Generate Pseudo-random Sequence::RC4 PRGA [C0021.004]                           |
| DATA                        | Compression Library:: [C0060]                                                   |
|                             | Encode Data::XOR [C0026.002]                                                    |
| DEFENSE EVASION             | Obfuscated Files or Information::Encoding-Standard Algorithm [E1027.m02]        |
| DISCOVERY                   | Application Window Discovery::Window Text [E1010.m01]                           |
|                             | Code Discovery::Enumerate PE Sections [B0046.001]                               |
| FILE SYSTEM                 | Create Directory:: [C0046]                                                      |
|                             | Delete Directory:: [C0048]                                                      |
|                             | Delete File:: [C0047]                                                           |
|                             | Get File Attributes:: [C0049]                                                   |
|                             | Move File:: [C0063]                                                             |
|                             | Read File:: [C0051]                                                             |
|                             | Set File Attributes:: [C0050]                                                   |
|                             | Writes File:: [C0052]                                                           |
| MEMORY                      | Allocate Memory:: [C0007]                                                       |
| OPERATING SYSTEM            | Registry::Create Registry Key [C0036.004]                                       |
|                             | Registry::Open Registry Key [C0036.003]                                         |
|                             | Registry::Query Registry Value [C0036.006]                                      |
| PROCESS                     | Create Mutex:: [C0042]                                                          |
|                             | Create Process:: [C0017]                                                        |
|                             | Create Thread:: [C0038]                                                         |
|                             | Resume Thread:: [C0054]                                                         |
|                             | Set Thread Local Storage Value:: [C0041]                                        |
|                             | Suspend Thread:: [C0055]                                                        |
+-----------------------------+---------------------------------------------------------------------------------+
```

A terceira e última tabela cita diretamente as capacidades encontradas de acordo com o *namespace* (categoria para um arquivo de regras YAML) que foi utilizado para

detectá-las. Como mostrado na Figura 3.19, é possível identificar capacidades conhecidas de bankers/infostealers.

Figura 3.19. Tabela de capacidades suspeitas e seu namespace de detecção

CAPABILITY	NAMESPACE
inspect load icon resource	anti-analysis
check for time delay via GetTickCount (3 matches)	anti-analysis/anti-debugging/debugger-detection
check for unmoving mouse cursor	anti-analysis/anti-vm/vm-detection
contain obfuscated stackstrings (2 matches)	anti-analysis/obfuscation/string/stackstring
get geographical location (4 matches)	collection
log keystrokes	collection/keylog
log keystrokes via polling (11 matches)	collection/keylog
capture screenshot	collection/screenshot
encode data using XOR (13 matches)	data-manipulation/encoding/xor
encrypt data using RC4 PRGA (3 matches)	data-manipulation/encryption/rc4
generate random numbers using the Delphi LCG	data-manipulation/prng/lcg

Utilizando a opção '-v' do CAPA, é possível identificar o endereço das funções suspeitas encontradas, conforme Figura 3.20:

Figura 3.20. Endereçamento de funções suspeitas identificadas pelo capa

```
log keystrokes
namespace  collection/keylog
scope      function
matches    0x55AF30

log keystrokes via polling (11 matches)
namespace  collection/keylog
scope      function
matches    0x4E6700
           0x4EA890
           0x4EEE40
           0x515DF8
           0x55AF30
           0x560898
           0x5608D4
           0x56BAA4
           0x56BAF8
           0x572AD4
           0x57E998

capture screenshot
namespace  collection/screenshot
scope      function
matches    0x4C4F14
```

Munido dessas informações, o analista poderá guiar melhor seus próximos passos nas análises posteriores. É possível até mesmo utilizar esse conhecimento efetuar operações, como alteração das chamadas de funções para evasão de virtualização e debugging, que podem atrapalhar as análises comportamentais e reversa.

3.6. Análise de Comportamento

Nesta seção, serão abordados os conceitos e as ferramentas necessárias para a análise manual de comportamento de artefatos suspeitos. Para tanto, serão utilizadas ferramentas de monitoramento de processos, de rede e de ações no sistema de arquivos e no registro do Windows.

O objetivo desta etapa da metodologia é examinar as ações realizadas pelo artefato suspeito quando executado. Diferentemente da análise estática, quando são observadas,

por exemplo, *strings* suspeitas apontando para um *hostname* desconhecido que podem indicar uma conexão maliciosa, na análise dinâmica ou de comportamento, poderemos observar de fato as funcionalidades do artefato analisado. No caso do exemplo da *string* suspeita, pode ser possível observar a tentativa de resolução do endereço do host suspeito, eventuais tentativas de conexão ao IP, qual porta foi utilizada, e assim por diante.

Como se pode observar, o objetivo é semelhante ao da análise automatizada com o uso de *Sandboxes*. As *Sandboxes* podem realmente ajudar na descoberta de comportametnos do artefato malicioso, no entanto, podem apresentar algumas limitações que poderão ser superadas na análise manual. Considere o seguinte exemplo: durante a análise de um artefato suspeito numa *Sandbox*, você identificou que o programa tenta realizar uma conexão na porta 80 de um determinado endereço IP. Como a funcionalidade da *Sandbox* é mais limitada e a tentativa de acesso falharia, você não conseguiria descobrir dados importantes que seriam transacionados na conexão.

Por outro lado, na análise manual de comportamento, dada a maior flexibilidade, esta limitação poderia ser superada. Para tanto, para o exemplo em questão, ao identificar que o artefato analisado estaria tentando uma conexão a um host da internet, você poderia simular que aquele endereço estaria disponível, permitindo que o malware estabelecesse uma conexão com um falso serviço enquanto você monitora os pacotes enviados.

Observe ainda neste exemplo que o acesso ao host desejado pelo malware foi simulado, ou seja, o malware não acessaria o host na internet, mas uma outra VM (no caso poderia ser a VM Linux), que seria preparada para aguardar a conexão do malware enquanto registraria todo o fluxo de dados.

Uma pergunta que você pode estar se fazendo é: por que não deixar o malware se comunicar com o host desejado na internet enquanto os pacotes são coletados? A resposta é que muito provavelmente você perderia o controle do processo de análise além de sinalizar ao atacante que o encontrou ou expondo a sua análise de alguma forma. Imagine se, neste caso, a conexão fosse para baixar e executar um novo conteúdo malicioso, que resultaria em novas conexões a outros endereços e novas ações no sistema operacional. Muito facilmente você poderia perder a trilha da análise em meio a uma grande quantidade de dados e ações.

Desta forma, o ideal é conduzir a análise de comportamento passo a passo, num ciclo contínuo de descoberta de recursos demandados pelo malware e fornecimento monitorado do recurso.

Nas subseçoes a seguir, apresentaremos a análise de comportamento do PE 'TextInputDocxs.exe'. Para isso, utilizaremos uma série de ferramentas que monitorarão o processo na busca por indicadores maliciosos tais como de alterações em arquivos, Registros do Windows e comunicação com endereços suspeitos (tráfego de comando e controle ou C2).

3.6.1. Ferramentas para análise de comportamento

A seguir a lista das ferramentas que serão utilizadas nesta análise:

- **Process Monitor**

O Proccess Monitor, também conhecido por 'procmon', é uma ferramenta avançada de monitoramento para o Windows que mostra em tempo real as atividades em arquivos, Registro do Windows e processos. Ela combina as funcionalidades de dois utilitários legados do Sysinternals, Filemon e Regmon, e adiciona uma extensiva lista de melhorias, incluindo filtragem de eventos, lista completa de propriedades de eventos tais como IDs de sessões e nomes de usuários, dados de processos, dados completos de threads com integração de suporte a símbolos para cada operação, registro de log simultâneo para um arquivo e outras mais. Por estes motivos, torna-se uma ferramenta fundamental na atividade de análise de comportamento de malwares.

O Proccess Monitor monitora todas as chamadas de sistema. Como há muitas chamadas de sistema no Windows (na casa de dezenas de milhares por minuto), executá-lo por muito tempo pode levar ao esgotamento dos recursos de RAM da máquina de análise. Desta forma, o ideal é executar a captura de eventos na ferramenta por um período curto de tempo - somente enquanto monitora as atividades do malware.

- **Process Hacker**

  Process Hacker é uma ferramenta gratuita e poderosa para monitoramento de processos e seus recursos, podendo ser bastante útil na análise de códigos maliciosos. A sua funcionalidade é semelhante ao gerenciador de tarefas do próprio Windows, com incrementos importantes. Como exemplo, podemos citar a possibilidade de descoberta de *handles* associados a um processo. *Handles* são apontadores para itens que foram abertos ou criados no sistema operacional, tais como janelas, processos, DLLs e arquivos. Esta funcionalidade pode ser útil, por exemplo, na análise de arquivos utilizados por um código malicioso. Uma outra funcionalidade que pode ser útil para a análise de comportamentos maliciosos é o monitoramento de rede. Com o Process Hacker é possível acompanhar conexões ativas por processo em tempo real.

- **Wireshark**

  Wireshark é uma das ferramentas mais utilizadas para análise de protocolos e tráfego de rede. A ferramenta, que é grátis e multiplataforma, permite a captura e a análise de centenas de protocolos. Do ponto de vista de análise de comportamento de um artefato suspeito, podemos dizer que é uma ferramenta fundamental uma vez que pode nos ajudar a entender como o malware está realizando comunicações de rede. Para o monitoramento do tráfego, utilizaremos o Wireshark na VM Linux capturando o tráfego gerado pelo artefato analisado na VM Windows.

- **FakeDNS**

  FakeDNS é uma ferramenta que simula um servidor DNS e responde um determinado endereço IP para qualquer consulta. Desenvolvida inicialmente por Francisco Santos [18], está disponível em uma versão modificada na distribuição REMnux. Do ponto de vista da análise de comportamento, esta ferramenta nos ajuda tanto na

[18]https://code.activestate.com/recipes/491264-mini-fake-dns-server

identificação de consultas DNS realizadas por um artefato suspeito quanto no direcionamento de eventuais conexões realizadas para um endereço IP de interesse do analista, onde a captura do tráfego estará sendo realizada. No nosso ambiente de análise, apontaremos o IP da VM Linux para qualquer consulta DNS realizada no na VM Windwows.

- **Noriben**

  Noriben [19] é um script em Python que funciona em conjunto com o Proccess Monitor para, automaticamente, coletar, analisar, e reportar indicadores de códigos maliciosos em tempo de execução. Em resumo, o Noriben faz com que o ambiente de execução do artefato suspeito, no nosso caso a VM Windodws, funcione de forma similar a uma Sandbox. Seu funcionamento, basicamente, consiste na sequinte sequencia de passos: executar o Noriben, executar o código malicioso e aguardar alguns instantes e depois interromper o Noriben. Ao final deste processo, estarão disponíveis um arquivo de texto com o report dos eventos identificados no período do monitoramento e um arquivo com extensão 'pml' da coleta realizada pelo Proccess Monitor.

- **ProcDOT**

  O ProcDOT [20] é uma ferramenta que contribui para a análise dos resultados obtidos pelo Proccess Monitor e pelo Wireshark (opcionalmente). Desenvolvida por Christian Wojner, do CERT da Austria, a ferramenta facilita a interpretação dos eventos coletados ao apresentá-los na forma de um grafo interativo. Desta forma, fica mais fácil de entender a relação entre o processo analisado e as ações realizadas ao longo do tempo, tais como a leitura de um registro do Windows e a gravação de um arquivo.

3.6.2. Análise de comportamento do binário TextInputDocxs.exe

Nesta seção apresentaremos a análise do binário TextInputDocxs.exe utilizando as ferramentas descritas no tópico anterior. Estamos utilizando este binário apenas como um exemplo. As mesmas técncias e ferramentas podem ser utilizadas na análise de outros artefatos.

3.6.2.1. Preparando o ambiente

Antes de dispararmos a execução do binário na VM Windows, precisaremos preparar o ambiente de monitoramento.

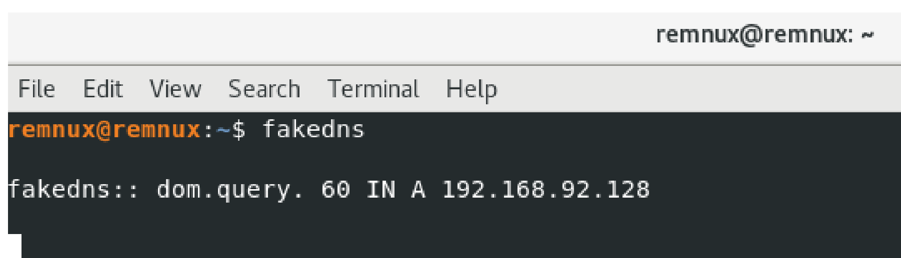
Para facilitar a condução da preparação e do processo de monitoramento, tome nota das interfaces de rede e dos endereços IP das interfaces de rede que conectam as VMs Windows e Linux. Estas interfaces estarão no modo 'Host Only'.

Preparando a VM Linux:

[19] https://github.com/Rurik/Noriben
[20] https://procdot.com/index.htm

- FakeDNS: no REMnux, abra um terminal de comando (menu Activities e depois Terminal) e digite o comando 'fakedns'. Isso fará com que um serviço falso de DNS passe a funcionar na VM Linux, conforme Figura 3.21.

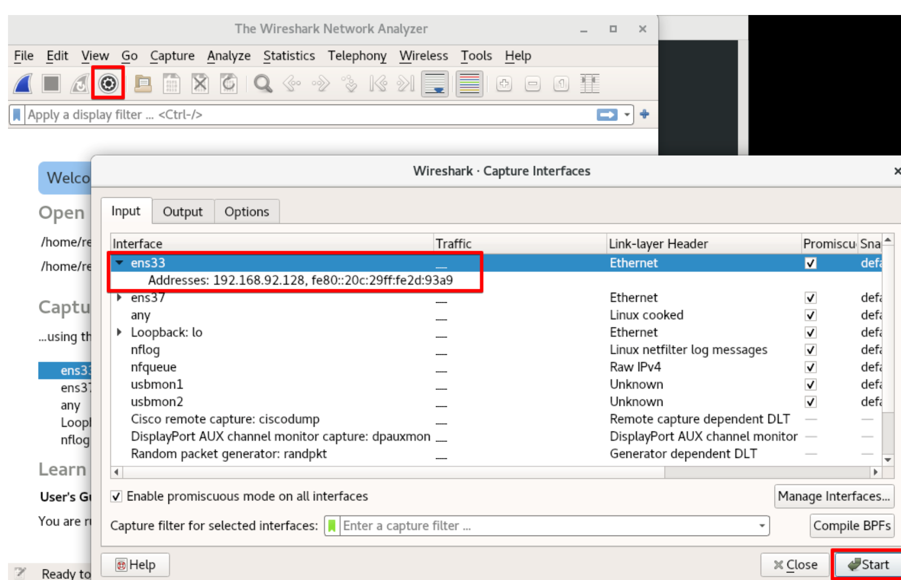
Figura 3.21. Fakedns em execução



- Wireshark: na sequencia, execute o Wireshark. Para isso, volte a clicar no botão 'Activities' e, na barra de busca qua aparecerá, digite 'Wireshark'. Clique no aplicativo 'Wireshark' resultante da busca. No Wireshark, clique no botão de monitoramento, selecione a interface que conecta a VM Linux a VM Windows e depois clique no botão 'Start', conforme Figura 3.22. Observe que, não necessariamente, o endereço IP da imagem corresponderá ao endereço IP utilizado na sua VM Linux. O importnate é que o monitorament seja feito na interface que conecta as duas VMs entre si.

  Por enquanto, deixe o Wireshark monitorando os pacotes. No entanto, antes de executarmos o malware, é recomendável que a captura seja reiniciada para termos um menor número de pacotes a analisar. O botão para reiniciar a captura está ao lado esquerdo do botão de início da captura na Figura 3.22.

Figura 3.22. Executando o Wireshark



Preparando a VM Windows:

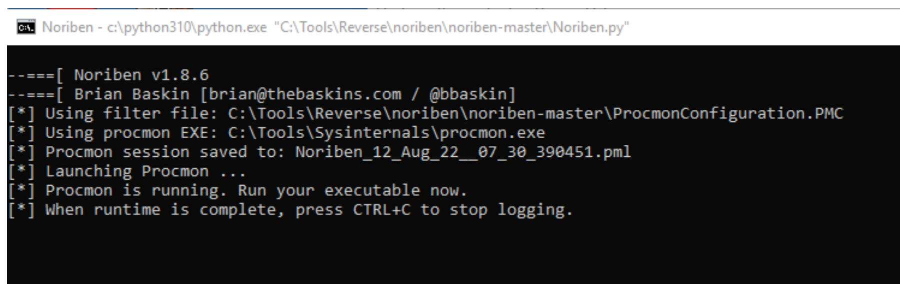
- Certifique-se de que a VM não esteja com acesso à Internet e que as configurações de Gateway e DNS da VM estejam apontando para o endereço IP da VM Linux. Para mais detalhes de como preparar o ambiente de laboratório, leia a seção 3.2.2.
- Execute o Process Hacker e o deixe em execução. Esta ferramenta será utilizada para monitorar o funcionamento do binário e para interrompê-lo quando for o momento certo;
- Binário suspeito: deixe o binário 'TextInputDocsx.exe' [21] em ponto de execução na VM Windows. O binário está no formato ZIP protegido por senha. A senha é 'infected'. É muito importante que só descompacte o arquivo dentro da VM Windows. Isso evitará que o antivirus do seu host possa removê-lo ou que você execute por acidente o binário no seu host;
- Noriben: localize um diretório de nome 'Noriben' na área de trabalho e abra-o. Neste diretório tem o script Noriben e o diretório de saída (output) dos resultados. Por enquanto, não execute ainda o script.
- Snapshot: com o ambiente preparado, crie um *snapshot* da VM Windows. Isso facilitará o retorno da VM exatamente ao estado em que está caso sejam necessárias outras execuções do malware - o que é muito comum de acontecer na análise de artefatos maliciosos.

3.6.2.2. Executando o código suspeito

Com o ambiente preparado, é hora de executar o código suspeito. O objetivo aqui é o monitoramento das ações do malware enquanto é executado. Para isso, siga os passos a seguir:

1. Volte a VM Linux e reinicie a captura do Wireshark;
2. Na VM Windows, execute o Noriben. Uma janela de Prompt de Comando será aberta neste momento, conforme 3.23. Conforme informado na tela, para finalizar o processo de monitoramento, deve-se digitar CTRL+C na tela do Noriben.

Figura 3.23. Noriben em execução



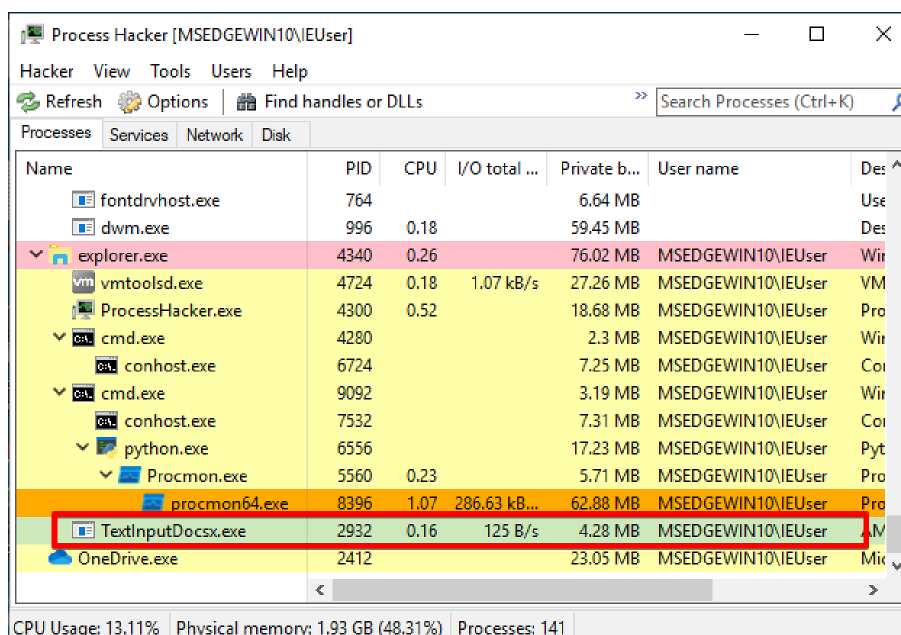
[21] https://github.com/rrmarinho/sbseg2022

3. Execute o binário suspeito: conforme orientações da seção anterior, o binário suspeito estará na VM Windows pronto para ser executado. Execute-o neste momento. A execução do binário suspeito não resultará em nenhuma nova janela. Será silenciosa.

4. Process Hacker: No Proccess Hacker, que já estará em funcionamento, localize o processo do binário suspeito, conforme Figura 3.24. Aguarde cerca de 30 a 40 segundos e interrompa o processo. Para isso, clique no processo com o botão direito e depois em 'Terminate'.

   O tempo a ser aguardado para interromper o processo não segue um padrão. Como esta é uma primeira execução, escolhemos interromper o processo com pouto tempo de execução para ver se algo já ocorreu. No entanto, pode acontecer de o malware ter sido programado para iniciar alguma ação somente após 1 min após ser executado ou até mais do que isso. O fato de não sabermos o tempo ideal a se aguardar no monitoramento é uma limitação da análise de comportamento que pode ser superada com a análises mais avançadas, com o uso de debuggers e disassemblers.

Figura 3.24. Processo TextInputDocsx.exe no Process Hacker

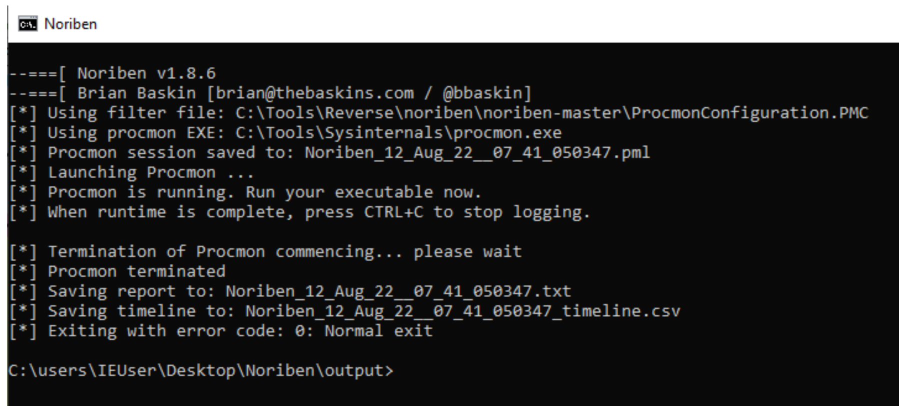


5. Encerrando o monitoramento no Noriben: volte a tela do Noriben, clique na tela preta do terminal para dar o foco e depois pressione CTRL+C. Com isso, o monitoramento será encerrado, conforme Figura 3.25.

   Ao mesmo tempo, uma janela de Bloco de Notas com o relatório de execução será aberta. Por enquanto, deixe o relatório aberto somente. Vamos tratar da análise no próximo tópico.

6. Wireshark: volte a VM Linux e interrompa o monitoramento do Wireshark usando o botão '*Stop Capture*' - o segundo da barra de ícones;

Figura 3.25. Encerrando o monitoramento do Noriben



7. FakeDNS: ainda na VM Linux, volte a janela do FakeDNS e interrompa o processo pressionando CTRL+C.

Uma vez encerrada a etapa de monitoramento, iniciaremos as análises.

3.6.2.3. Analisando os resultados

Até aqui, preparamos o ambiente, executamos o artefato suspeito enquanto o monitorávamos e agora chegou a etapa de análise dos resultados. Confira a seguir os resultados em cada uma das ferramentas:

1. **Noriben**

   O relatório do Noriben é dividido em 5 seções principais: *Process Created*, *File Activity*, *Registry Activity*, *Network Activity* e *Unique Hosts*. Em cada uma destas seções, estarão presentes todos os eventos capturados pelo Proccess Monitor enquanto esteve em execução. Desta forma, é importante que tenha em mente que os eventos apresentados no relatório incluem os eventos causados pelo binário executado, mas não somente. Vão ser apresentados eventos diversos que ocorreram no sistema operacional no período da coleta. É exatamente por isso que o quanto mais restrito for o período da coleta, menor será a quantidade de eventos a serem analisados.

   Após uma análise do relatório do Noriben, observamos que o processo 'TextInputDocsx.exe', cujo PID (Process ID) era 2932, criou o arquivo '%AppData% \Processtime.html', conforme Figura 3.26.

   Um próximo passo da análise seria a abertura do arquivo para verificar seu conteúdo, por exemplo. Neste caso, o arquvio está vazio e, por isso, não acrescenta muito. No entanto, de qualquer forma, a criação do arquivo em si pode ser considerada um indicador de comprometimento (IOC) deste artefato. Isso quer dizer que, se você tiver que identificar computadores onde este mesmo artefato tenha sido executado, a identificação da presença deste arquivo poderia ser um excelente indício.

Figura 3.26. Relatório do Noriben

```
Noriben_12_Aug_22__07_41_050347.txt - Notepad
File  Edit  Format  View  Help
-=] Sandbox Analysis Report generated by Noriben v1.8.6
-=] Developed by Brian Baskin: brian @@ thebaskins.com  @bbaskin
-=] The latest release can be found at https://github.com/Rurik/Noriben

-=] Execution time: 50.22 seconds
-=] Processing time: 2.05 seconds
-=] Analysis time: 2.09 seconds

Processes Created:
==================
[CreateProcess] Explorer.EXE:4340 > "%UserProfile%\Desktop\SAMPLES-CURSO-v4\SAMPLES-CURSO\TextInputDocsx.exe "  [Child PID: 2932]
[CreateProcess] svchost.exe:772 > "%WinDir%\system32\DllHost.exe /Processid:{E10F6C3A-F1AE-4ADC-AA9D-2FE65525666E}"     [Child PID: 3816]
[CreateProcess] svchost.exe:772 > "%WinDir%\system32\DllHost.exe /Processid:{E10F6C3A-F1AE-4ADC-AA9D-2FE65525666E}"     [Child PID: 7868]

File Activity:
==================
[CreateFile] TextInputDocsx.exe:2932 > %AppData%\Progrestime.html       [SHA256: e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855]
[CreateFile] svchost.exe:1592 > %WinDir%\ServiceProfiles\LocalService\AppData\Local\FontCache   [File no longer exists]
[CreateFile] svchost.exe:1592 > %WinDir%\ServiceProfiles\LocalService\AppData\Local\FontCache   [File no longer exists]
[RenameFile] svchost.exe:1592 > %WinDir%\ServiceProfiles\LocalService\AppData\Local\FontCache\FontCache-FontSet-S-1-5-18.dat => %WinDir%\ServicePrc
[RenameFile] svchost.exe:1592 > %WinDir%\ServiceProfiles\LocalService\AppData\Local\FontCache\FontCache-S-1-5-18.dat => %WinDir%\ServiceProfiles\Lc
[CreateFile] svchost.exe:1592 > %WinDir%\ServiceProfiles\LocalService\AppData\Local\FontCache\~FontCache-S-1-5-18.dat   [File no longer exists]
```

2. **FakeDNS**

   Observando as resoluções de DNS realizadas pelo FakeDNS, identificamos duas que potencialmente estão ligadas à execução do código suspeito, conforme 3.27. Análises posteriores nos ajudarão a identificar se foram ou não originadas pelo binário. Por enquanto há um bom indício pois são as únicas que diferem das outras que foram consultas DNS a domínios Microsoft - provavelmente realizadas por ferramentas do SO.

Figura 3.27. Análise do FakeDNS

```
remnux@remnux: ~
File  Edit  View  Search  Terminal  Help
remnux@remnux:~$ fakedns

fakedns:: dom.query. 60 IN A 192.168.92.128

Response: disc501.prod.do.dsp.mp.microsoft.com -> 192.168.92.128

Response: disc501.prod.do.dsp.mp.microsoft.com -> 192.168.92.128

Response: livekernelreports.duckdns.org -> 192.168.92.128

Response: disc501.prod.do.dsp.mp.microsoft.com -> 192.168.92.128

Response: freedow.ml -> 192.168.92.128
```

3. **Wireshark**

   De volta ao Wireshark, identificamos uma sequencia de conexões bastante interessante, conforme pode ser visto na Figura 3.28. Acompanhe os itens a seguir para entender os detalhes nas referidas linhas indicados na figura.

   - Linhas 88 e 96: requisições DNS para resoluções dos nomes livekernelreports.duckdns.org e freedow.ml - os mesmos indicados no FakeDNS;
   - Linhas 89 e 97: respostas do FakeDNS devolvendo o IP 192.168.92.128 à consulta DNS, o que é esperado;

- Linhas 90 e 98: pacotes de início de conexão (SYN) para as portas TCP/80 e TCP/26457 do IP 192.168.92.128. Esta conexão, originalmente, seria destinada aos endereços IP resultantes das consultas DNS acima. Isso indica que o malware poderia estar tentando conexão com o atacante. Observe que as conexões não tiveram sucesso uma vez que estas portas não estão abertas no host 192.168.92.128 - que, neste caso é a VM Linux.

Figura 3.28. Análise do Wireshark

No.	Time	Source	Destination	Protocol	Length	Info
85	13.730597107	192.168.92.1	224.0.0.251	MDNS	87	Standard query 0x0000 PTR _spotify-connect._tcp.local, "QM" question
86	13.732259256	fe80::c892:842b:a6a…	ff02::fb	MDNS	107	Standard query 0x0000 PTR _spotify-connect._tcp.local, "QM" question
87	17.576619921	192.168.92.1	239.255.255.250	SSDP	167	M-SEARCH * HTTP/1.1
88	22.635939649	192.168.92.132	192.168.92.128	DNS	89	Standard query 0xc6b1 A livekernelreports.duckdns.org
89	22.636224757	192.168.92.128	192.168.92.132	DNS	105	Standard query response 0xc6b1 A livekernelreports.duckdns.org A 192.168.92.128
90	22.642531940	192.168.92.132	192.168.92.128	TCP	66	50212 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
91	22.642558640	192.168.92.128	192.168.92.132	TCP	54	80 → 50212 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
92	23.155026626	192.168.92.132	192.168.92.128	TCP	66	[TCP Retransmission] 50212 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_
93	23.155054526	192.168.92.128	192.168.92.132	TCP	54	80 → 50212 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
94	23.671814212	192.168.92.132	192.168.92.128	TCP	66	[TCP Retransmission] 50212 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_
95	23.671842513	192.168.92.128	192.168.92.132	TCP	54	80 → 50212 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
96	24.724112602	192.168.92.132	192.168.92.128	DNS	70	Standard query 0x1ac0 A freedow.ml
97	24.724433312	192.168.92.128	192.168.92.132	DNS	86	Standard query response 0x1ac0 A freedow.ml A 192.168.92.128
98	24.726699578	192.168.92.132	192.168.92.128	TCP	66	50213 → 26457 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
99	24.726728678	192.168.92.128	192.168.92.132	TCP	54	26457 → 50213 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
100	25.264875759	192.168.92.132	192.168.92.128	TCP	66	[TCP Retransmission] 50213 → 26457 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SA
101	25.264913760	192.168.92.128	192.168.92.132	TCP	54	26457 → 50213 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
102	25.758749430	192.168.92.132	192.168.92.128	TCP	66	[TCP Retransmission] 50213 → 26457 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SA
103	25.758776831	192.168.92.128	192.168.92.132	TCP	54	26457 → 50213 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
104	25.763430266	192.168.92.132	192.168.92.128	TCP	66	50214 → 26457 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1

```
▸ Frame 1: 212 bytes on wire (1696 bits), 212 bytes captured (1696 bits) on interface ens33, id 0
▸ Ethernet II, Src: VMware_c0:00:01 (00:50:56:c0:00:01), Dst: IPv4mcast_7f:ff:fa (01:00:5e:7f:ff:fa)
▸ Internet Protocol Version 4, Src: 192.168.92.1, Dst: 239.255.255.250
▸ User Datagram Protocol, Src Port: 61692, Dst Port: 1900
▸ Simple Service Discovery Protocol
```

4. **ProcDOT**

   De volta à VM Windows, execute o software ProcDOT. O ícone deve estar na área de trabalho.

   Na primeira vez que é executado, o ProcDOT precisa ser configurado. Na janela de opções, informe os caminhos das ferramentas solicitas:

   - Path to windump/tcpdump: C:\textbackslash Tools \Reverse\windump \WinDump.exe
   - Path to dot (Graphviz): C:\Program Files\Graphviz\bin\dot.exe

   Uma vez configurado, no campo 'Procmon' da tela, informe o path do arquivo no formato 'csv' gerado pelo Noriben e no campo 'Launcher', busque pelo processo 'TextInputDocsx.exe'. Siga os passos de 1 a 4 na Figura 3.29.

   Uma vez selecionados os parâmetros (não esqueça de clicar em Refresh), um grafo será apresentado pelo ProcDOT representando os eventos relacionados ao processo selecionado, conforme Figura 3.30.

   Observe que este resultado resume bem o comportamento do artefato até aqui: a criação do arquivo Progresstime.html e as tentativas de conexão nas portas TCP/80 e TCP/26457 ao endereço IP da VM Linux. Aqui comprovamos a suspeita que foi levantada lá na análise do FakeDNS e do Wireshark - as conexões às portas suspeitas foram realmente feitas pelo binário analisado. Uma outra forma de comprovar isso seria através do Process Hacker.

Figura 3.29. Seleção de parâmetros no ProcDOT

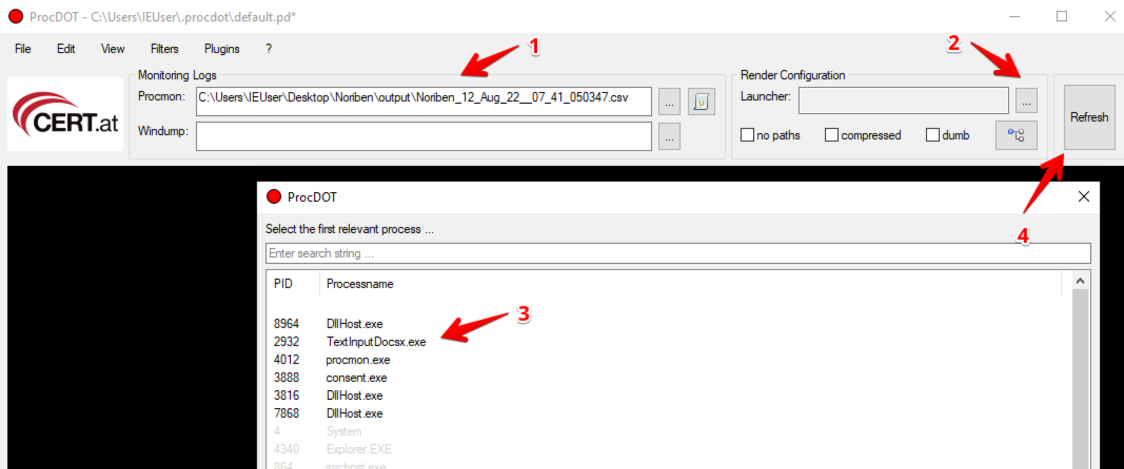
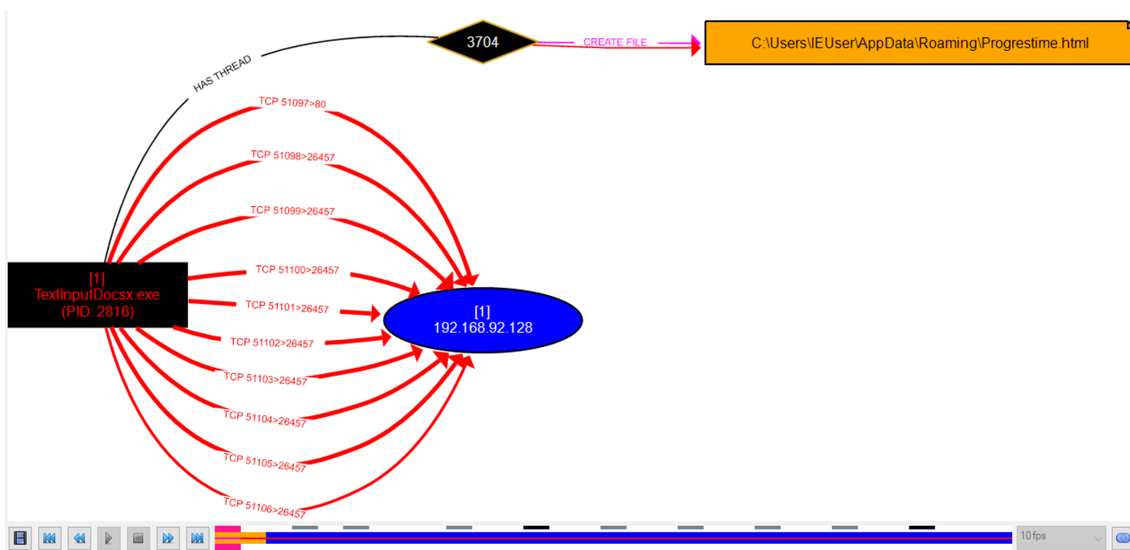


Figura 3.30. Resultado ProcDOT



3.6.2.4. Próximos passos

Até aqui, observamos alguns indicadores de comprometimento ligados ao binário analisado:

1. Criação do arquivo '%AppData% \Processtime.html'.
2. Consultas DNS aos nomes: livekernelreports.duckdns.org e freedow.ml;
3. Tentativas deconexões TCP nas portas 80 e 23457;

No entanto, podemos tentar avançar mais na análise de comportamento. Conforme

explicado na introdução desta seção, na análise de comportamento podemos fornecer de forma gradativa os recursos demandados pelo artefato.

Um próximo passo natural para esta análise seria tentar descobrir o que o malware estaria buscando nas portas às quais tentou se conectar. Para tanto, abriremos as portas demandadas na VM Linux e aguardaremos a conexão enquanto monitoramos o tráfego via Wireshark.

Para fornecer as portas 80 e 23457, proceda da seguinte forma na VM Linux:

- Porta 80: no prompt de comando da VM Linux, inicie o serviço http. Na tela de comando, digite 'httpd start' e <ENTER>. Com isso, um serviço HTTP server será iniciado na VM Linux;
- Porta 23457: para este serviço, iniciaremos um socket TCP utilizando o Netcat. Para isso, execute no terminal o comando 'nc -l -p 26457'.

Agora, refaça o experimento seguindo os seguintes passos:

1. Retorne o *snapshot* da VM Windows. Isso vai garantir que observaremos a primiera infecção do malware na máquina. Opcionalmente, para este caso em específico, basta apagar o arquivo '%AppData% \Processtime.html' que o malware realizará todo o comportamento;
2. Na VM Linux, certifique-se de deixar o FakeDNS e Wireshark ativos;
3. Inicie o 'TextInputDocsx.exe' na VM Linux enquanto monitora o tráfego no Wireshark.

Como resultado, você deve observar algo semelhante ao que temos na Figura 3.31.

Figura 3.31. Resultado Wireshark

```
 4 17.534969266 192.168.92.1    192.168.92.255 UDP   86 57621 → 57621 Len=44
 5 17.909340821 192.168.92.132  192.168.92.128 DNS   89 Standard query 0x1a26 A livekernelreports.duckdns.org
 6 17.909553021 192.168.92.128  192.168.92.132 DNS  105 Standard query response 0x1a26 A livekernelreports.duckdns.org A 192.168.92.128
 7 17.914854536 192.168.92.132  192.168.92.128 TCP   66 51225 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
 8 17.919832050 192.168.92.128  192.168.92.132 TCP   66 80 → 51225 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM=1 WS=128
 9 17.920330451 192.168.92.132  192.168.92.128 TCP   60 51225 → 80 [ACK] Seq=1 Ack=1 Win=262656 Len=0
10 17.920689152 192.168.92.132  192.168.92.128 HTTP 290 POST /ups/ HTTP/1.1
11 17.920737352 192.168.92.128  192.168.92.132 TCP   54 80 → 51225 [ACK] Seq=1 Ack=237 Win=64128 Len=0
12 17.920928553 192.168.92.128  192.168.92.132 HTTP 402 HTTP/1.1 405 Not Allowed  (text/html)
13 17.961490668 192.168.92.132  192.168.92.128 TCP   60 51225 → 80 [ACK] Seq=237 Ack=349 Win=262400 Len=0
14 18.964813984 192.168.92.132  192.168.92.128 DNS   70 Standard query 0xe7f5 A freedow.ml
15 18.965016385 192.168.92.128  192.168.92.132 DNS   86 Standard query response 0xe7f5 A freedow.ml A 192.168.92.128
16 18.971348102 192.168.92.132  192.168.92.128 TCP   66 51226 → 26457 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
17 18.971445302 192.168.92.128  192.168.92.132 TCP   66 26457 → 51226 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM=1 WS=128
18 18.971968504 192.168.92.132  192.168.92.128 TCP   60 51226 → 26457 [ACK] Seq=1 Ack=1 Win=2102272 Len=0
19 19.964065155 192.168.92.132  192.168.92.128 TCP   79 51226 → 26457 [PSH, ACK] Seq=1 Ack=1 Win=2102272 Len=25
20 19.964124055 192.168.92.128  192.168.92.132 TCP   54 26457 → 51226 [ACK] Seq=1 Ack=26 Win=64256 Len=0
```

Observe agora que as conexões nas portas 80 e 26457 foram estabelecidas com sucesso. Adicionalmente, descobrimos que na porta TCP/80 o malware enviou um HTTP request do tipo POST na URL '/ups/'. Podemos, inclusive descobrir mais indicadores, como, por exemplo o 'user-agent' utilizado pelo malware na requisição HTTP. Para isso, basta inspecionar a conexão na porta 80 no Wireshark com o botão direito, depois Follow e depois TCP Stream. O resultado será algo semelhante ao da Figura 3.32.

Figura 3.32. Resultado Wireshark - HTTP Request

Wireshark · Follow TCP Stream (tcp.stream eq 0) · ens33

```
POST /ups/ HTTP/1.1
Connection: Keep-Alive
Content-Type: application/x-www-form-urlencoded
Accept: */*
User-Agent: Mozilla/4.0 (compatible; Win32; WinHttp.WinHttpRequest.5)
Content-Length: 0
Host: livekernelreports.duckdns.org

HTTP/1.1 405 Not Allowed
Server: nginx/1.14.0 (Ubuntu)
Date: Fri, 12 Aug 2022 18:36:02 GMT
Content-Type: text/html
Content-Length: 182
Connection: keep-alive
```

Para a conexão TCP/26457, a mesma análise pode ser feita. Adicionalmente, na própria tela onde executamos o Netcat, veremos o resultado. Observe que o malware realizou uma conexão na porta aberta com o Netcat e enviou uma sequencia de dados 'LOGIN TestUser password'. Muito provavelmente, está enviando informações para o atacante ou tentando autenticar a nova vítima. Quando estiver com tempo, tente interagir com o malware por meio do Netcat e veja se ele responde.

3.7. Análise reversa

A análise reversa, também chamada de engenharia reversa, é a última etapa de análise, sendo a mais custosa em questão de tempo e complexa em nível de conhecimento. Nela, o analista utilizará ferramentas do tipo *debugger* ou *disassembler*, que permitirão realizar procedimentos de alteração de fluxo de código, modificação de regiões da memória RAM e de registradores em um dado processo [Wong 2018],[Dang et al. 2014].

Alguns procedimentos, como descriptografia de conteúdo e identificação e bypass de técnicas anti-sandbox só podem ser realizadas, depedendo do cenário, com uma análise dinâmica profunda. É nela também que será possível entender exatamente o funcionamento de cada thread ou função importante executando no processo.

3.7.1. Arquitetura de computadores

A arquitetura de computadores pode ser pensada como uma organização dos componentes físicos e virtuais que fornece abstrações para o sistema operacional. As arquiteturas de computadores modernos são baseados na arquitetura de Von Neumann, que definiu componentes como a *Control Processing Unit (CPU)*, *Control Unit*, *Instruction Register*, e a memória principal (RAM) para armazenamento de dados e instruções. Seu funcionamento utiliza alguns registradores, que são memórias extremamente voláteis embutidas no processador, para realizar o ciclo de *fetch-decode-execute*, que busca uma instrução na RAM para ser executada, a decodifica e executa.

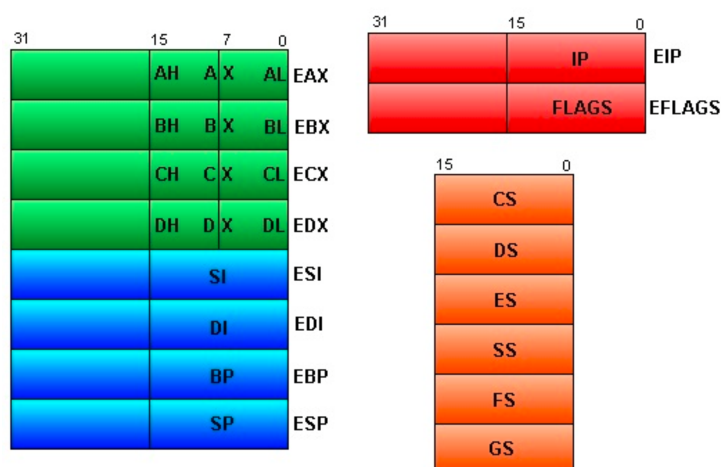
As arquiteturas são categorizadas de acordo com o tamanho da palavra, ou seja, quantos bits o processador consegue processar em um ciclo de execução. Isso também limita a quantidade de memória principal máxima que um processador pode gerenciar, visto que um registrador específico é utilizado nas arquiteturas modernas para apontar o próximo endereço da RAM que deverá ser decodificado e executado.

As arquiteturas modernas geralmente tem o tamanho de uma palavra de 32-bit ou 64-bit e contém uma série de registradores para uso geral, que auxiliam na execução de um programa. Exemplos de arquiteturas são a x86, amd64 e ARM, e cada um tem definido uma *Instruction Set Architecture (ISA)*, que é o conjunto de instruções que são disponibilizadas para o sistema operacional, além de um conjunto de registradores para propósitos gerais, segmentação e controle de fluxo. Entre os registradores mais importantes na arquitetura x86, estão:

- **Propósito geral:** EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP
- **Controle de fluxo:** EIP, EFLAGS
- **Segmentos**: CS, DS, ES, SS, FS, GS

Na Figura 3.33, é possível ter uma visão desses registradores com relação a quantidade de bits máxima que pode ser armazenada. Note que alguns registradores de 32-bits, como EAX, EBX, ECX e EDX podem ser acessados pelas suas representações de 16-bits (AX, BX, CX, DX), a parte mais significativa dos 16-bits (AH, BH, CH, DH) e a parte menos significativa dos 16-bits (AL, BL, CL, DL). Esse acesso é possível devido a retrocompatibilidade com programas escritos para arquiteturas antigas da Intel, que contavam com palavras de 16-bits e 8-bits.

Figura 3.33. Registradores x86 e a quantidade de bits que armazenam



Os registradores EIP e EFLAGS são registradores especiais, pois o EIP irá indicar o endereço da próxima instrução a ser executada, enquanto que o EFLAGS é uma máscara de bits que cada bit irá representar uma flag de comparação. Essas flags são configuradas

de acordo com o resultado da última instrução executada, como, por exemplo, se a instrução retornou que o valor de dois registradores é igual. Na listagem abaixo, estão algumas flags que podem ser configuradas nesse registrador:

- **ZF**: a Zero Flag é ativada quando o resultado de uma operação é igual a zero. Caso contrário, é desativada;
- **CF**: a Carry Flag é ativada quando o resultado de uma operação é muito grande ou muito pequeno para o operando de destino. Caso contrário, é desativada;
- **SF**: a Sign Flag é ativada quando o resultado de uma operação é negativo. Caso contrário, é desativada;
- **TF**: a Trap Flag é usada para debug. O processador x86 executará uma instrução por vez se esta flag estiver ativada;

O sistema operacional utiliza essas abstrações para criar mais abstrações de código, como *system calls* e *APIs*, e de estruturas de dados para gerenciamento de memória. Entre elas, estão:

- **stack**: área utilizada para variáveis locais e parâmetros para funções;
- **heap**: área utilizada para alocação dinâmica de memória durante a execução do programa;

Durante o carregamento de um código, a stack é uma estrutura de dados que irá crescer dos endereços mais altos para os endereços mais baixos. Isso quer dizer que ao diminuirmos um endereço na stack, estamos alocando regiões novas de memória para ela. A heap, por sua vez, cresce de endereços menores para os endereços maiores, em direção à stack. Além disso, a stack é gerenciada por dois registradores: EBP e ESP, responsáveis por armazenar o endereço da base da pilha e do topo dela, respectivamente, e por operações de empilhamento e desempilhamento.

3.7.2. Assembly x86

Nesta seção serão abordados aspectos teóricos que servirão de base para a engenharia reversa de binários PE.

Antes do tratamento do assunto Assembly, é importante relembrar o processo de produção de um executável. Para a construção de um arquivo executável pelo sistema operacional, é geralmente utilizada uma linguagem de alto nível, como C/C++, para a escrita do código-fonte. Esse código-fonte é tratado por um compilador, que, depois de diversas etapas, como análise léxica, sintática e montagem, gerará um código de máquina que poderá ser lido pelo processador. Um exemplo de código-fonte escrito na linguagem C está exemplificado na Figura 3.34.

O código de máquina gerado é composto de zeros e uns, em uma ordem lógica de opcodes e argumentos. Os opcodes são números inteiros que podem ser entendidos

Figura 3.34. Exemplo de código-fonte escrito na linguagem C

```
10
11  int main(int argc, char** argv) {
12
13      int a = 10;
14      int b = 20;
15      int c = soma(a,b);
16      printf("A soma eh: %d", c);
17
18      return 0;
```

como o endereço de instruções dentro do *ISA*, e são geralmente representados pelo seu valor em hexadecimal. Alguns opcodes aceitam operandos, que são bytes que deverão estar imediatamente após eles, e o conjunto opcode e operando forma uma **instrução**. Tais operandos podem ser registradores, endereços de memória ou valores imediatos (constantes inteiras). Os primeiros 16 bytes gerados pela compilação do programa mostrado na Figura 3.34 estão demonstrados em sua forma hexadecimal e binária na Figura 3.35.

Figura 3.35. Exemplo de código de máquina gerado por um programa escrito em C

```
55 89 E5 83 E4 F0 83 EC 20 E8 98 09 00 00 C7 44...

1010101100010011110010110000011111001001111100001000
00111110110000100000111010001001100000001...
```

Programar utilizando apenas código de máquina é uma tarefa desafiadora, propensa a erros. Para isso, foi criado o que é chamado de linguagem Assembly, ou linguagem de montagem, que é uma tradução das instruções do código de máquina para palavras, também chamados de *mnemônicos* e seus operandos, que é mais legível para seres humanos. A partir de um arquivo com instruções escritas em Assembly, é possível gerar o código de máquina por um processo de montagem, ao se utilizar um software *assembler*.

Como Assembly é uma tradução direta dos opcodes e argumentos escritos em código de máquina, é possível construir duas sintaxes de linguagens de montagem para um mesmo conjunto de instruções. Para a arquitetura x86, existem duas muito utilizadas: a notação Intel e a ATT. A notação Intel é comumente utilizada em contexto de engenharia reversa em Windows, enquanto que a ATT é mais utilizada no contexto Linux.

Segue abaixo um exemplo da sintaxe Intel. Nela, a ordem dos operandos é ao contrário: 'MOV ECX, AABBCCDDh' significa colocar o valor imediato 'AABBCCDD' no registrador 'ECX', não é necessário sufixar o mnemônico da operação e a derreferência de um ponteiro é feita com o uso de colchetes.

```
mov ecx, AABBCCDDh
mov ecx, [eax]
```

Em contraste com a sintaxe Intel, abaixo está um exemplo de uso da sintaxe ATT. Nela, a ordem dos operandos é da esquerda para direita: 'movl $0xAABBCCDD, %ecx'

significa colocar o valor imediato 'AABBCCDD' no registrador ECX. Note que os registradores são prefixados com o símbolo '%', o mnemônico da operação é sufixado para indicar o tamanho do operando ('movl' indica operandos de 32-bits) e a derreferência de um ponteiro é feita com o uso de parênteses.

```
movl $0xAABBCCDD, %ecx
movl (%eax), %ecx
```

Na listagem abaixo, temos algumas das principais operações que um analista de malware irá se deparar no cotidiano. Para uma lista completa, é recomendado checar o site da Intel[22] para um melhor direcionamento na documentação.

- MOV, LEA: utilizados para movimentação de valores e passagem de ponteiros, respectivamente;
- PUSH, POP: operações de empilhamento e desempilhamento, respectivamente, realizadas na stack;
- ADD, SUB, MUL, DIV: operações aritméticas de soma, subtração, multiplicação e divisão;
- OR, AND, XOR, NEG, NOT: operações lógicas bit-a-bit de ou, e, xor, complemento de 1 e complemento de 2;
- JMP, CALL, RET: mudança do fluxo de código através de mudança incondicional, chamada e retorno de funções/procedimentos;
- CMP, TEST: utilizados para comparação, configuram o valor do registrador EFLAGS de acordo com a comparação realizada.

O processo de converter o código de máquina para Assembly, conhecido como disassembly, também é possível e é um dos recursos que facilita a inspeção a baixo nível de um código compilado. Esse procedimento é realizado através de ferramentas *disassembler*, e permitirá uma análise estática a nível de código de máquina, mais aprofundada do que a análise estática básica realizada nas seções anteriores. A Figura 3.36 representa o processo de disassembly, com o código de máquina à esquerda e sua representação Assembly à direita.

A Figura 3.37 representa, em alto nível, o processo de compilação e disassembly.

É importante destacar que binários escritos em linguagens que geram códigos gerenciados (não nativos) em *intermediate languages*, como Python e Java, que usam máquinas virtuais para interpretar seus códigos, ou que contenham um compilador *Just In Time (JIT)* embutido, como C#, terão instruções que não são específicas para uma arquitetura real de computadores. Ou seja, utilizam instruções que não serão interpretadas por um processador real, e sim um virtual, que irá realizar o procedimento de tradução dessas instruções para as instruções da arquitetura na qual o programa está sendo executado. Como exemplo desse tipo de instrução, na Figura 3.38 está exemplificado as instruções necessárias para chamar a função *print* em Python:

[22]https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Figura 3.36. Exemplo do disassembly de um programa escrito em C

```
55                     | push ebp
89 E5                  | mov ebp,esp
83 E4 F0               | and esp,FFFFFFF0
83 EC 20               | sub esp,20
E8 98 09 00 00         | call exemplo1.401EC0
C7 44 24 1C 0A 00 00   | mov dword ptr ss:[esp+1C],A
C7 44 24 18 14 00 00   | mov dword ptr ss:[esp+18],14
8B 44 24 18            | mov eax,dword ptr ss:[esp+18]
89 44 24 04            | mov dword ptr ss:[esp+4],eax
8B 44 24 1C            | mov eax,dword ptr ss:[esp+1C]
89 04 24               | mov dword ptr ss:[esp],eax
E8 B4 FF FF FF         | call exemplo1.401500
```

Figura 3.37. Processo de compilação e disassembly

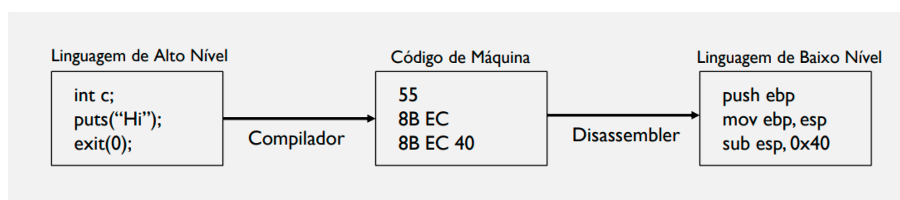


Figura 3.38. Exemplo do código interpretado pela máquina virtual python

```
1  print('Hello World')

1  1     0 LOAD_NAME        0 (print)
2        2 LOAD_CONST       0 ('Hello World')
3        4 CALL_FUNCTION    1
4        6 POP_TOP
5        8 LOAD_CONST       1 (None)
6       10 RETURN_VALUE
```

3.7.3. Descompilação

Outra forma de realizar análise estática de código é através do processo de descompilação, realizada por um *decompiler*, no qual há a tentativa de reconstrução do código-fonte original a partir do código de máquina. Esse processo gera dados muito importantes para uma rápida análise do fluxo de código de um artefato, porém essa reconstrução não retornará o código-fonte original de um programa nativo, devido a perdas de dados resultantes do processo de compilação.

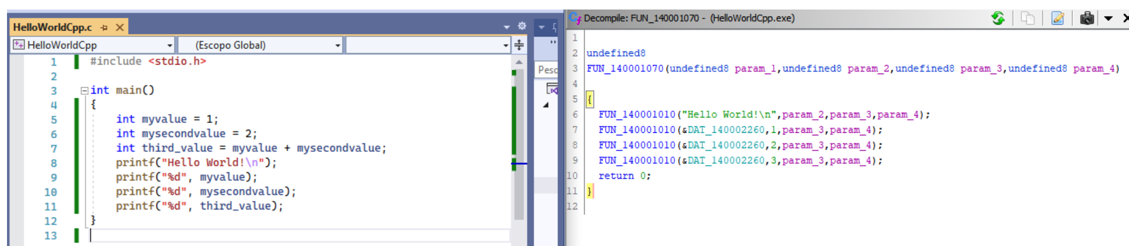
Nomes de variáveis locais são um exemplo de um desses dados perdidos e que não poderão ser retornados pelo processo de descompilação. Um exemplo de perda está demonstrado na Figura 3.39, em que o programa original escrito em C, à esquerda, contém os nomes das variáveis locais, enquanto que o mesmo programa descompilado pela ferramenta *Ghidra*[23] não possui, além dos nomes das funções que não estão mais presentes na versão descompilada.

Uma exceção à essas perdas significativas de informação são binários com código gerenciado. Devido a necessidade de um maior controle sobre variáveis e outros componentes de um código, é necessário que o binário gerado tenha metadados que um programa nativo não dispõe. Através desses metadados, é posssível gerar um descompilador mais efetivo que consegue encontrar informações de classes, namespaces e até nome de variáveis.

Para exemplificar esse conceito, na Figura 3.40 está explicitado, à esquerda, o

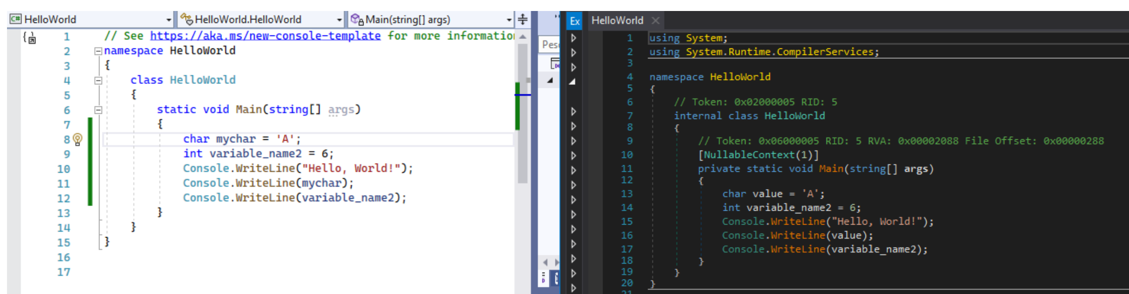
[23]https://ghidra-sre.org/

Figura 3.39. Processo de descompilação de um código nativo



código-fonte original de um programa simples escrito em C#, e na direita o seu código descompilado a partir da ferramenta *dnSpy*[24]. A diferença para um código nativo é notável, visto que o nome do namespace, da classe e até das variáveis locais foram restaurados com sucesso.

Figura 3.40. Processo de descompilação de um código gerenciado



3.7.4. Debugging

Essa etapa compreenderá a execução de um EXE em um processo debugger. Ela será importante para entender o passo-a-passo da execução de uma ou mais threads durante a execução do processo.

Na listagem abaixo, estão descritas algumas funções implementadas nos principais debuggers atuais:

- **step into**: prossegue a execução para a próxima instrução, inclusive para instruções dentro de funções, e pausa o processo;
- **step over**: prossegue a execução para a próxima instrução, executando todas as instruções dentro de uma determinada função, retornando para a função que chamou e pausando o processo;
- **step out**: caso o registrador de instrução esteja dentro de uma função, ele irá executar todas as instruções daquela instrução até o fim dela.

[24]https://github.com/dnSpy/dnSpy

Outra funcionalidade de debuggers importante para engenharia reversa são os *breakpoints*. Eles são tipos de interrupção que permitem a um *debugger* pausar a execução de um processo. Existem três tipos de breakpoints: de software, de memória e de hardware.

O breakpoint de software é representado na arquitetura x86 pela instrução *INT 3*, ou interrupção 3, que irá sinalizar para o *debugger* que o processo deverá ser parado e analisado. Quando o processador encontra essa instrução, ele irá sinalizar para o sistema operacional, que por sua vez irá sinalizar para o *debugger* que o processo foi pausado, permitindo realizar análises de memória e de código.

O breakpoint de hardware é dependente da arquitetura e do processador, pois são registradores físicos que identificam a condição para a parada do processo, e não uma instrução por si só, e por isso não são sobrescritos por operações, como os de software. Podem ser configurados breakpoints de hardware para leitura, escrita e execução. Seu principal uso é para identificar o momento de execução de um determinado byte que tem seu valor alterado constantemente, ou reside em uma região de memória alocada dinamicamente.

O breakpoint de memória é utilizado quando se sabe o intervalo de páginas de memória em que determinada execução de código poderá acontecer, mas não um endereço em específico. Ele é configurado a partir da mudança das permissões de páginas de memória para que qualquer leitura, escrita ou execução gere uma exceção a ser tratada pelo debugger. Ele, por sua vez, irá parar a execução para que o analista consiga dar prosseguimento ao processo de análise.

Existem diversas ferramentas que dão acesso a funcionalidades de debugging, sendo um dos mais conhecidos e usados em análise de malware a ferramenta x64dbg, ou x32dbg no caso de binários 32-bits. Na Figura 3.41, é possível perceber a tela principal do programa quando um binário é carregado a partir do menu **File -> Open**.

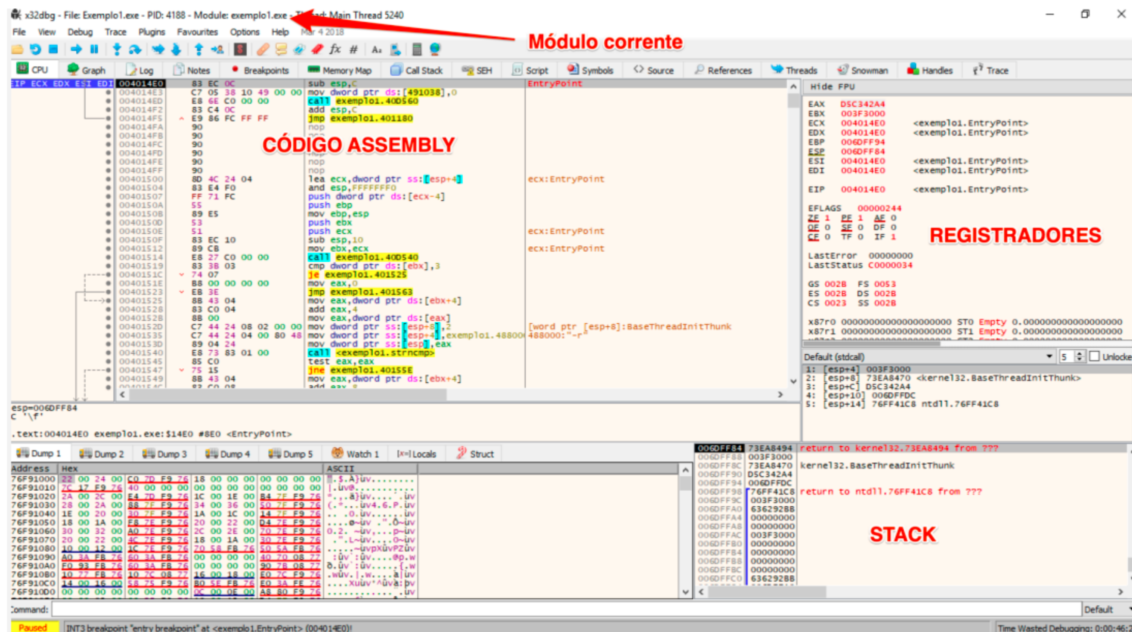
Alguns pontos importantes sobre essa janela:

- módulo corrente: informa para o analista qual módulo atual ele está analisando;
- código assembly: disassembly do módulo ou de outras partes;
- registradores: valores atuais dos registradores;
- stack: valores atualmente armazenados na stack de execução.

Utilizando os conhecimentos de análise reversa, vamos buscar dar um passo a mais na análise do nosso binário 'TextInputDocsx.exe'. A análise reversa, como visto nesta seção, é a etapa mais complexa e a que pode levar mais tempo do analista. O mais indicado, portanto, é mirar pontos específicos de interesse do código ao invés de fazer uma análise passo a passo de todo o binário.

Neste sentido, para esta análise reversa, escolhemos buscar um entendimento melhor do ponto onde paramos na análise de comportamento. Vimos que há uma comunicação do malware com a porta TCP/26457 na qual verificamos o envio de uma mensagem

Figura 3.41. Layout de tela principal do x64dbg



'LOGIN TestUser password'. Para simular o lado do servidor, utilizamos a ferramenta NetCat (comando: nc -l -p 26457).

Com o mesmo ambiente utilizado na análise de comportamento (incluindo o NetCat), executando o binário na ferramenta x32dbg, siga os passos listados a seguir. Eles servirão de guia na busca de um melhor entendimento da comunicação do malware com o servidor de comando e controle (C2) e de exemplo de como a ferramenta pode ser utilizada em futuras análises.

1. Execute o binário no x32dbg. Logo que iniciar o *debugging*, você perceberá que estará no módulo 'ntdll.dll'. Clique no botão 'run' para avançar para o 'EntryPoint' do código do binário a ser analisado;

2. Com o NetCat aguardando a conexão, execute o 'run' mais uma vez. Você deverá perceber que a mensagem 'LOGIN TestUser password' chegou no NetCat e que o código no x32dbg está no modo 'Running' (informação no canto inferior esquerdo);

3. Agora teremos que identificar a parte do código que está executando a comunicação com a porta 26457 e aguardando uma resposta. Para isso, mude para a aba 'Threads' e identifique a *thread* 'Main'. Ela estará com o valor 'UserRequest' no campo 'Wait Reason'. Clique 2x na *thread* e depois mude para a aba 'Call Stack'. Na aba 'Call Stack' é apresentada a lista de instruções chamadas com as mais recentes primeiro. Observe que a segunda instrução chama uma função da biblioteca mswsock, uma biblioteca responsável por comunicação de rede. Como o nosso interesse é monitorar o código que chama a função de comunicação, vamos inserir um *breakpoint* na mswsock. Para isso, selecione a linha e depois pressione F2. Você perceberá que um *breakpoint* foi inserido na aba 'Breakpoints';

4. Agora, na tela que está executando o NetCat, digite um conteúdo qualquer e depois digite <ENTER>;

5. Neste momento, o x32dbg deve estar parado na instrução imediatamente após a chamada à instrução 'mswsock.7464A460'. Clique em 'run' novamente;

6. De volta a tela do NetCat, você poderá observar que um novo conteúdo foi enviado pelo malware (<|ARQUIVO1|>AMIGO1<|>Windows 10 (Version 10.0, Build 17763, 64-bit Edition)<|>MSEDGEWIN10<|>);

7. Agora, experimente enviar um novo conteúdo na tela do NetCat em resposta ao malware. (Ex: novoconteudo). Até aqui, praticamente repetimos o que fizemos na análise de comportamento utilizando uma ferramenta de *debugging*, o que nos dá mais controle;

8. Agora, volte ao x32dbg para dar continuidade na análise do que será feito com a novo conteúdo enviado ao malware. Um atalho importante neste momento é o botão 'Run to user code' que fica na barra de ferramentas. Com ele, você avançará no restante das instruções da biblioteca mswsock, que não nos interessam, e vamos até a próxima instrução no código do nosso binário. O debugger avançará até a instrução '00598B8C' do nosso binário;

9. Agora, utilizando a função 'Step Over' ou F8, avance nas instruções até chegar na de endereço '005C1A69'. A partir deste endereço, teremos instruções que compararão o conteúdo digitado com uma string decodificada em tempo de execução. Observação importante: tive que seguir as instruções uma a uma até chegar em algo que me chamasse a atenção. Caso queira avançar diretamente para a instrução '005C1A69', pode optar por configurar um *breakpoint* nela e depois mandar executar o código com o 'run'. Para isso, utilizando a barra inferior de comandos do x32dbg digite o comando: setbpx 0x005C1A69 e depois <ENTER>;

10. Continue a execução das instruções com 'Step Over' até chegar na instrução '005C1AAD'. Neste ponto, observe que a execução da instrução anterior decodificou o conteúdo 'DOWNLOAD1';

11. Analisando as instruções seguintes, verificamos que a função chamada na instrução '005C1AB3' faz uma comparação do 'novoconteudo' com 'DOWNLOAD1'. Este pode ser um sinal importante para a nossa análise. Execute novamente os passos até aqui e forneça o conteúdo 'DOWNLOAD1' ao malware ao invés de 'novoconteudo';

12. Dando continuidade a análise, a comparação feita na função em '005C1AB3' agora será verdadeira. Na sequencia, na instrução '005C1AD2' será solicitada uma nova entrada no NetCat. Para facilitar o retorno à análise depois do fornecimento do novo conteúdo, inclua um *breakpoint* na instrução '005C1ADA';

13. No NetCat, digite o comando '1'. OBS: experimentei outros conteúdos de texto e uma exceção foi gerada porque a entrada não foi numérica;

14. Ao retornar ao x32dbg, execute o 'run' para retomar a partir do *breakpoint*. Na sequencia, na instrução '005C1B1B' um novo input será solicitado. Informe '1' também e, da mesma forma, configure um *breakpoint* para retornar ao código com facilitade. Neste caso, pode ser configurado para a instrução '005C1B28';

15. Nas instruções subsequentes, observe que há uma nova decodificação de string. Neste caso, será decodificada a string '.zip' após a execução da instrução '005C1B96';

16. Na sequencia, ao chegar na instrução '005C1BBB' utilizando 'Step Over', você poderá verificar que um path de arquivo estará na pilha (*stack*). No meu caso, o path é C:\Users\IEUser\AppData\Roaming\Microsoft\p1E8u4t5r2.zip. No entando, a cada execução, o nome do arquivo muda e o path se mantém. Inclusive, você poderá observar que o arquivo 'zip' foi criado;

17. Avançando nas instruções com 'Step Over', notei uma execção ao passar pela chamada no endereço '005C1C93'. Notei que isso ocorreu pelo fato de o arquvio 'zip' estar vazio. É possível que o arquivo seja criado por algum outro caminho que não o que seguimos. Para evitar a exceção e continuar a análise, preparei um zip com um conteúdo qualquer para sobrescrever o zip solicitado pelo malware. Faça o mesmo e reinicie o processo até aqui substituindo o seu zip pelo criado pelo malware antes da execução da instrução '005C1C93';

18. Na nova execução, você passará pela instrução '005C1C93' sem a exceção. Avance até a instrução '005C1D5A' e observe que um novo conteúdo foi decodificado: 'RtkNGUI.exe';

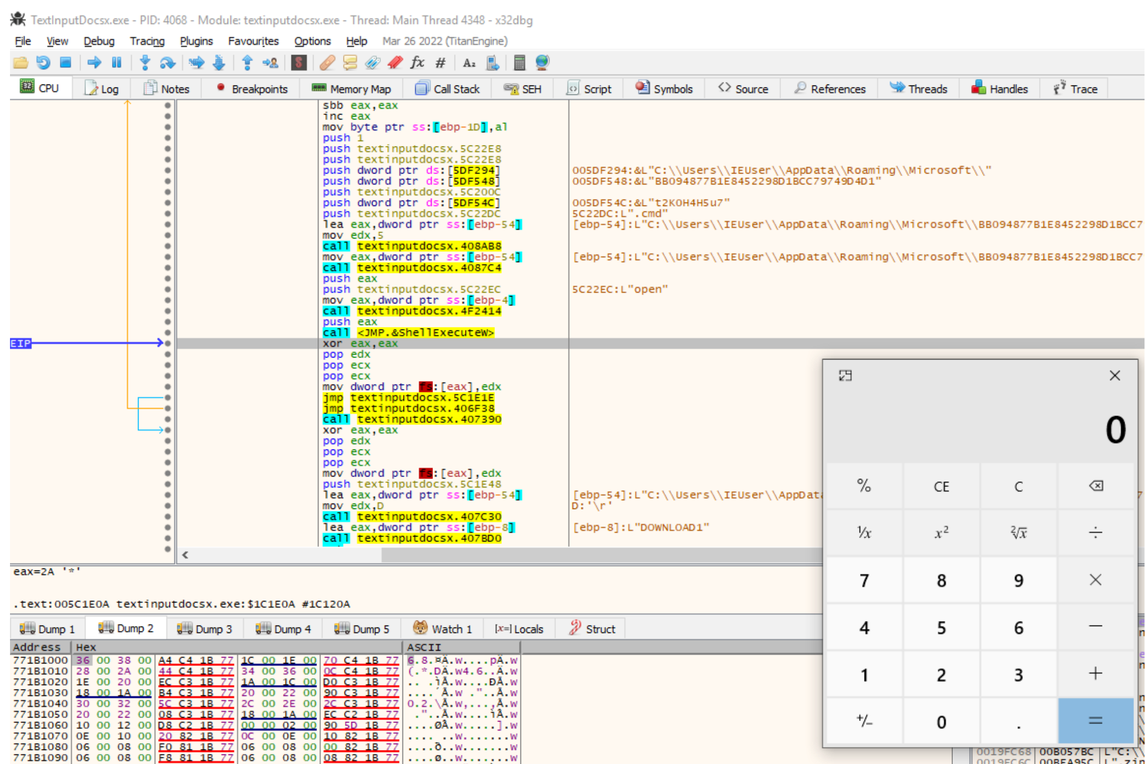
19. Avançando nas instruções com 'Step Over', pare na de endereço '005C1DAB', onde uma chamada para a função 'MoveFileW' é executada. Observe que, neste momento, o código renomeará o arquivo 'RtkNGUI.exe' para 'RtkNGUI.cmd'. Ou seja, ele espera que dentro do zip exista um arquivo de nome 'RtkNGUI.exe'. Neste momento, recriei o zip contendo o binário da calculadora do Windows renomeada para 'RtkNGUI.exe' e refiz o processo. Faça o mesmo;

20. Avance com 'Step Over' até a instrução '005C1E05', onde será executada uma chamada para a função 'ShellExecuteW'. Observe nos parâmetros para a função que o arquivo 'RtkNGUI.cmd' será executado;

21. Avance mais um passo com o 'Step Over' e veja que a calculadora será executada, como na Figura 3.42.

    Esta sequencia exemplifica como uma análise reversa pode ser realizada para buscar um maior entendimento do comportamento de artefatos maliciosos com o uso de um *debugger*. Até aqui, verificamos que o código executa um conteúdo adicional contido em um arquivo zip. Outras descobertas podem ser feitas com análises semelhantes.

Figura 3.42. Execução de comando pelo malware



3.8. Conclusão

Nesse minicurso apresentamos os conceitos básicos de análise de códigos maliciosos e introduzimos o leitor nas características particulares das soluções de análise para o ambiente *Windows*.

Acreditamos que a partir dos conhecimentos obtidos neste curso, os leitores estarão aptos a realizar procedimentos básicos de análise de malware, em especial nas seguintes áreas: *Profiling* do artefato suspeito (OSINT), análise automatizada, análise estática, análise comportamento e análise reversa. Esperamos, assim, contribuir para o desenvolvimento das pesquisas dos leitores em análise de códigos maliciosos de maneira geral.

Por fim, com o propósito de obter uma melhor compreensão dos temas abordados ao longo deste capítulo e apresentados no minicurso, recomendamos ao leitor aprofundar os conhecimentos por meio da leitura dos trabalhos referenciados.

Agradecimentos. Os autores agradecem à empresa Morphus Segurança da Informação por meio do Morphus Labs e ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), em especial via Programa de Bolsas de Produtividade em Pesquisa (processo número 306389/2020-7).

Referências

[Andriesse 2018] Andriesse, D. (2018). *Practical Binary Analysis*. No Starch Press.

[Bianco 2022] Bianco, D. (2022). Pirâmide da dor. http://www.sans.org/tools/the-pyramid-of-pain/.

[Botacin et al. 2018] Botacin, M. F., Geus, P. L., and Grégio, A. R. A. (2018). The other guys: automated analysis of marginalized malware. *Journal of Computer Virology and Hacking Techniques*, 14(1):87–98.

[Carvey 2014] Carvey, H. (2014). *Windows Forensic Analysis Toolkit*. Syngress.

[Dang et al. 2014] Dang, B., Gazet, A., Bachaalany, E., and Josse, S. (2014). *Practical Reverse Engineering*. Wiley.

[Galante et al. 2018] Galante, L., Botacin, M., Grégio, A., and de Geus, P. L. (2018). Malicious linux binaries: A landscape. *Anais Estendidos do XVIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*, pages 213–222.

[Gragido 2012] Gragido, W. (2012). Understanding indicators of compromise (ioc) part i. *Dostupné z http://blogs. rsa. com/understanding-indicators-of-compromise-ioc-part-i/Ověřeno ke dni*, 18(5):2016.

[Kim 2018] Kim, P. (2018). *The Hacker Playbook 3: Practical Guide To Penetration Testing*. Independently published.

[Kleymenov and Thabet 2019] Kleymenov, A. and Thabet, A. (2019). *Mastering Malware Analysis*. Packt Publishing Ltd.

[Liu et al. 2022] Liu, S., Feng, P., Wang, S., Sun, K., and Cao, J. (2022). Enhancing malware analysis sandboxes with emulated user behavior. *Computers & Security*, 115:102613.

[Mills and Legg 2020] Mills, A. and Legg, P. (2020). Investigating anti-evasion malware triggers using automated sandbox reconfiguration techniques. *Journal of Cybersecurity and Privacy*, 1(1):19–39.

[Monnappa 2018] Monnappa, K. A. (2018). *Learning Malware Analysis: Explore the concepts, tools, and techniques to analyze and investigate Windows malware*. Packt Publishing Ltd.

[Sahay et al. 2020] Sahay, S. K., Sharma, A., and Rathore, H. (2020). Evolution of malware and its detection techniques. In Tuba, M., Akashe, S., and Joshi, A., editors, *Information and Communication Technology for Sustainable Development*, pages 139–150, Singapore. Springer Singapore.

[Sikorski 2012] Sikorski, Michael; Honig, A. (2012). *Practical malware analysis: the hands on guide to dissecting malicious software*. no starch press.

[Skoudis and Zeltser 2003] Skoudis, E. and Zeltser, L. (2003). *Malware: Fighting Malicious Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[Wong 2018] Wong, R. (2018). *Mastering Reverse Engineering: Re-engineer your ethical hacking skills*. Packt Publishing Ltd.

© 2022 Sociedade Brasileira de Computação