Módulo de análise forense de RAM para Volatility 3.0

Mestrado em Cibersegurança e Informática Forense

Leonardo Dias da Silva

Trabalho de Projeto realizado sob a orientação do Professor Luís Alexandre Lopes Frazão

Leiria, novembro de 2021

Originalidade e Direitos de Autor

O presente relatório de projeto é original, elaborado unicamente para este fim, tendo sido devidamente citados todos os autores cujos estudos e publicações contribuíram para o elaborar.

Reproduções parciais deste documento serão autorizadas na condição de que seja mencionado o Autor e feita referência ao ciclo de estudos no âmbito do qual o mesmo foi realizado, a saber, Curso de Mestrado em Cibersegurança e Informática Forense, no ano letivo 2020/2021, da Escola Superior de Tecnologia e Gestão do Instituto Politécnico de Leiria, Portugal, e, bem assim, à data das provas públicas que visaram a avaliação destes trabalhos.

Agradecimentos

Agradeço ao Professor Luís Frazão pela orientação desde a conceptualização deste projeto até à sua conclusão sem a qual teria sido impossível do realizar.

Ao meu colega Vítor Manuel Sabino Morais pela contribuição na conceptualização inicial deste projeto.

Aos meus amigos, um grande agradecimento pelo apoio.

Resumo

O número de dispositivos conectados à Internet está cada vez mais a aumentar, com isto também o número de ataques nesta vertente tem aumentado bastante. Existe cada vez mais a necessidade de proceder a algum tipo de análise aos sistemas digitais que todos nós usamos no nosso dia-a-dia e para tal temos a análise forense digital.

Nos dispositivos temos diversos componentes e sistemas internos que podem ser alvo de uma análise forense digital, um dos componentes que tem ganho cada vez mais relevância é a análise de memória RAM. As análises a discos de armazenamento já era algo comum, no entanto muito *malware* atualmente acaba por esconder ou eliminar este tipo de informação após execução. Algum *malware* consegue até executar apenas na RAM o que torna a análise desta memória essencial para uma análise forense.

Uma das ferramentas mais populares para este tipo de análise é o Volatility que contém diversos módulos que permitem a análise de RAM. Algo também bastante usado são as regras Yara, que permitem efetuar pesquisas na RAM, em ficheiros de *crash*, ou até mesmo em imagens de máquinas virtuais. É possível utilizar as regras Yara juntamente com o Volatility, uma vez que existe um módulo no Volatility que permite esta utilização, sendo este módulo denominado yarascan.

No entanto o yarascan tem algumas limitações. Este projeto é desenvolveu um módulo que mitiga algumas dessas limitações, sendo a principal que apenas é possível correr um ficheiro de regras Yara com cada execução do yarascan.

Este módulo desenvolvido pretende também melhorar a forma como o resultado é apresentado ao utilizador, sendo que para tal vai ser apresentado em formato CSV. O módulo criado tem também como objetivo interagir com o GitHub de forma a disponibilizar alguns repositórios de regras Yara ao utilizador.

Os resultados obtidos com este projeto permitem verificar que o módulo criado é realmente mais eficiente quando existem diversos ficheiros de regras Yara e que os resultados são apresentados de uma forma mais amigável para o utilizador.

Palavras-chave: Volatility, yarascan, regras Yara, malware, análise forense digital

Abstract

The number of connected devices to the Internet is rising every time, which makes the number of attacks to this kind of systems is rising as well. There is a growing need to have an analysis to the attacked systems that we use every day, for this we have digital forensics.

In the devices we have multiple components and internal systems that can be analysed, one of the components that has been gaining relevance in forensics is the RAM. Analysis to storage drives have been common for a while, however malware in the current days hides either hides the changes they do in the storage or delete them after execution. Some malware can execute just by using the RAM which makes the analysis of this memory essential in digital forensics.

One of the most popular tools for this analysis is Volatility, this tool has multiple modules that allow to analyse RAM memory. Something that is also highly used is Yara rules, these rules give us the ability to do searches in the content of the RAM, crash files or images from virtual machines. It is possible to use Yara rules along with Volatility, since there is a module called yarascan in Volatility that gives us this ability.

The module yarascan has some limitations. This project developed a module that mitigates some of these limitations, the principal one being that yarascan can only run a Yara rules file each time. The goal with this module is to run multiple files instead of just one.

The created module also aims to improve the output that is shown to the user, the output will not only be shown on screen but also sent to CSV files. This module will also interact with GitHub with the objective to make some Yara rules repositories available to the user.

The results obtained with this project show that the module created is much more efficient to use than yarascan when there are multiple files to run, and the results are shown in a more user-friendly way.

Keywords: Volatility, yarascan, Yara rules, malware, digital forensics

Índice

Originalidade e Direitos de Autor ..... **iii**

Agradecimentos ..... **iv**

Resumo ..... **v**

Abstract ..... **vii**

Índice ..... **ix**

Lista de Figuras ..... **xi**

Lista de Tabelas ..... **xii**

Lista de siglas e acrónimos ..... **xiii**

1. Introdução **1**

1.1. Objetivos e Contribuições **2**

1.2. Estrutura do Trabalho **4**

2. Estado da Arte **5**

2.1. RAM **5**

2.2. Análise Forense Digital **5**

2.2.1. Análise Forense na RAM 6

2.3. Yara Rules **7**

2.4. Software **9**

2.4.1. Volatility 10

2.4.2. IDA Pro 10

2.4.3. Cuckoo Sandbox 10

2.5. Python **11**

2.6. Problema **11**

2.7. Síntese **12**

3. Solução Proposta **13**

3.1. Arquitetura da Solução **13**

3.2. Atualização dos repositórios Yara **15**

3.3. Repositórios GitHub de Regras Yara **15**

3.3.1. Yara-Rules Yara Project ... 16
3.3.2. ESET Malware-IoC ... 16
3.3.3. ReversingLabs Yara Rules ... 16
3.3.4. Bartblaze Yara-Rules ... 16
3.3.5. Signature-Base ... 17
3.3.6. McAfee Yara-Rules ... 17
3.3.7. Yara-Forensics ... 17
3.3.8. Citizen Lab malware-signatures ... 17
3.3.9. AlienVaultLabs Malware Analysis ... 18

3.4. Síntese ... 20

4. Desenvolvimento da Solução ... 21

4.1. Bibliotecas Python ... 21

4.2. Volatility Framework ... 21

4.3. Requisitos para criar um módulo para o Volatility ... 28

4.4. Instalação do módulo ... 30

4.5. Arquitetura Solução Final ... 31

4.6. Cuckoo Sandbox ... 38

4.7. Síntese ... 39

5. Testes e resultados ... 41

5.1. Testes de eficiência ... 41

5.2. Testes Multiplataforma ... 44
5.2.1. Testes sistema Windows ... 44
5.2.2. Testes sistema Linux ... 46

5.3. Síntese ... 49

6. Conclusão ... 51

6.1. Trabalho Futuro ... 52

Referências Bibliográficas ... 53

Lista de Figuras

Figura 1 Arquitetura do conceito inicial 13
Figura 2 Exemplo de relatório Cukoo Sandbox [19] 14
Figura 3 Exemplo de execução do comando pslist 22
Figura 4 Exemplo de execução do pstree 23
Figura 5 Execução do comando dlllist 24
Figura 6 Exemplo de execução do comando certificates 25
Figura 7 Exemplo de execução do comando HiveList 26
Figura 8 Exemplo de execução do comando Info 27
Figura 9 Resultado execução do comando yarascan 28
Figura 10 Arquitetura Final da Solução 31
Figura 11 Exemplo execução da opção rules 32
Figura 12 Exemplo de execução da opção category 32
Figura 13 Fluxo da análise de RAM através do módulo 33
Figura 14 Fluxos dos comandos do GitHub 34
Figura 15 Resultados dos comandos pelo método tradicional 42
Figura 16 Resultados da execução do módulo em formato CSV 43
Figura 17 Execução do comando rules na máquina Windows 45
Figura 18 Ficheiros com resultados do comando rules na máquina Ubuntu 45
Figura 19 Execução do comando clone na máquina Windows 46
Figura 20 Estrutura de ficheiros criada pelo comando clone na máquina Windows 46
Figura 21 Execução de comando rules em Ubuntu 47
Figura 22 Ficheiros com resultados do comando rules na máquina Ubuntu 47
Figura 23 Execução do comando clone na máquina Ubuntu 47
Figura 24 Estrutura de ficheiros criada pelo comando clone na máquina Ubuntu 48
Figura 25 Comparação entre o comando em máquinas Windows e Linux 48

Lista de Tabelas

Tabela 1 Cronograma das tarefas macro deste projeto 3

Tabela 2 Comparativo de repositórios GitHub com Regras Yara 19

Tabela 3 Comparação em segundos para analisar diversas quantidades de ficheiros de regras 43

Lista de siglas e acrónimos

API	Application Programmable Interface
DLL	Dynamic-link Library
DDoS	Distributed Denial of Service
IoC	Indicators of Compromise
IoT	Internet of Things
IPS	Intrusion Prevention System
NCCIC	National Cybersecurity and Communications Integration Center
PIP	Preferred Installer Program
RAM	Random-access memory

1. Introdução

Este trabalho é focado na análise forense digital, especificamente na análise forense de *Random-access memory*(RAM). A análise forense digital é cada vez mais importante devido ao número crescente de dispositivos existentes. Existem estimativas que em 2025 existirá cerca de 75 mil milhões de dispositivos ligados à internet [1]. Este número crescente de dispositivos traz preocupações grandes a nível de segurança, uma vez que grande parte destes dispositivos serão parte da *Internet of Things* (IoT), em 2021 são cerca de 12,3 mil milhões de dispositivos deste tipo conectados [2]. O problema com este tipo de dispositivos é que por vezes a sua capacidade de processamento não é suficiente para implementar segurança sobre as funções que já tem de implementar, ou por vezes é descorada por quem implementa o *software* nestes dispositivos.

Devido a este grande número de dispositivos em que a segurança pode ser mínima ou até mesmo nula, existe a capacidade de criar ataques de *Distributed Denial of Service* (DDoS), basicamente este tipo de ataques consiste em enviar pedidos falsos para uma máquina ou um domínio, estes pedidos podem ser feitos através de dispositivos IoT.

Um exemplo para um ataque deste tipo foi o ataque da Mirai *botnet* em 2016 [3], que foi um dos maiores ataques de DDoS que conseguiu desativar grande parte da internet dos Estados Unidos da América. A Mirai *botnet* era constituída principalmente por dispositivos IoT como camaras digitais. Existe uma estimativa que esta *botnet* tinha cerca de cem mil pontos maliciosos e existem relatórios que dizem que o ataque atingiu 1,2 *terabits* por segundo [3].

Devido a este tipo de ataques e outros que são mais invasivos com intuito de roubo de informação ou de efetuar um pedido de resgate sobre a informação que foi cifrada através de *malware,* é cada vez mais essencial fazer análises aos equipamentos digitais para eventualmente conseguirmos provas que nos levem aos autores dos ataques e à eventual utilização em tribunal para os condenar.

Dentro da análise forense digital temos a análise de memória RAM que tem ganho cada vez mais importância devido a muito *malware* tentar esconder a sua presença ao máximo no sistema de ficheiros, mas mesmo este tipo de *malware* tem de fazer uso da RAM para executar as suas funções, e é na RAM que podemos encontrar coisas importantes como

ficheiros executados, ligações ao exterior, os processos que correu, e até mesmo chaves secretas que sejam usadas para cifragem de conteúdo.

É numa das ferramentas mais utilizadas na análise de RAM, que é o Volatility, que este projeto se vai focar criando um módulo para esta ferramenta que pode facilitar o uso por parte de pessoas inexperientes e automatizar alguns processos.

1.1. Objetivos e Contribuições

Este projeto tem como objetivo principal desenvolver um módulo para a ferramenta Volatility. Este módulo tem como objetivos automatizar e simplificar o uso de regras Yara. Com isto queremos também apresentar os resultados do uso destas regras através do Volatility de uma forma mais prática para o utilizador.

Este módulo também tem como objetivo o uso de regras a partir de repositórios GitHub que possam de alguma forma ajudar na análise de uma imagem de memória RAM.

Com este módulo queremos desenvolver algo que combata a limitação atual do comando yarascan em executar apenas um ficheiro, e melhorar a forma como apresenta os conteúdos ao utilizador e tornar a sua utilização mais simples.

Para desenvolver este módulo foi necessário efetuar alguma pesquisa de forma a perceber como se poderia criar um módulo para esta ferramenta e fazer uso de um módulo nativo, neste caso o yarascan. Após obtermos os requisitos básicos módulo Volatility que fizesse uso do módulo nativo yarascan, foi desenvolvido um módulo que tivesse a capacidade de ler vários ficheiros de regras Yara, e apresentar o seu resultado em ficheiros de Excel. No final foi feita a introdução dos repositórios GitHub e de comandos para fazer uso deles e gerir o seu *download* e atualização.

Na Tabela 1 em formato de cronograma, é apresentado em formato temporal as tarefas macro realizadas neste projeto.

Tabela 1 Cronograma das tarefas macro deste projeto

Tarefas	2020				2021										
	9	10	11	12	1	2	3	4	5	6	7	8	9	10	11
Idealização conceito Inicial															
Pesquisa sobre Cuckoo															
Pesquisa como desenvolver um módulo para o Volatility															
Pesquisa sobre repositórios de regras Yara															
Implementação Cuckoo Sandbox															
Testes com Cuckoo															
Desenvolvimento de módulo Volatility															
Elaboração Relatório															

1.2. Estrutura do Trabalho

No capítulo 2 serão apresentadas as diversas tecnologias e conceitos envolvidos na realização deste projeto, tal como é o caso da RAM, da análise forense, e outras ferramentas existentes para além do Volatility que são usadas para análise forense de RAM.

Durante o capítulo 3 é demonstrada uma solução que foi idealizada no início, são descritos em mais detalhe os objetivos deste módulo e como se pretende que o mesmo funcione. São apresentados os diversos repositórios de regras Yara que foram considerados para a utilização neste módulo e uma avaliação dos mesmos.

No capítulo 4 é feita uma introdução à linguagem usada para o desenvolvimento deste módulo e a sua importância, demonstram-se diversos comandos que podem ser usados no Volatility de forma a dar um panorama geral do que esta ferramenta nos permite fazer. São apresentados os requisitos mínimos para se começar a desenvolver um módulo Volatility e os requisitos para efetuar a instalação deste módulo em específico. Por fim é apresentada uma arquitetura final da solução e as diversas opções que esta inclui, apresentando fluxos de execução deste módulo e apresentado o código das diversas opções. São também mencionados alguns problemas que ocorreram e decisões tomadas.

No decorrer do capítulo 5 são mencionados os diversos testes que foram efetuados com este módulo em alguns cenários diferentes. É também mencionada a participação num concurso sobre este tipo de módulos.

No capítulo final é apresentada uma breve conclusão do trabalho desenvolvido e apresentado o trabalho futuro.

2. Estado da Arte

Este capítulo faz o enquadramento da importância da análise de RAM dentro de uma análise digital forense. São apresentadas algumas das vantagens e de informação importante que pode ser retirada da mesma, mas também alguns desafios que esta análise apresenta. São abordadas metodologias de análise forense e o funcionamento da RAM de uma forma a dar uma visão geral do que é possível. Aborda-se algumas ferramentas que auxiliam este tipo de análise e alguns dos seus constrangimentos e desafios na sua utilização por pessoas menos experientes.

2.1. RAM

RAM é um tipo de memória que pode ser lida e alterada, usada para quaisquer dados e processos que estejam em execução no sistema. A RAM permite que dados possam ser lidos ou escritos na mesma quantidade de tempo independente da sua localização física no dispositivo.

Quando a RAM contém dados que já não são necessários não os apaga logo, apenas marca o espaço como livre e assim quando for necessário aquele espaço vai ser reescrito com algo novo. Portanto, podem ser retirados dados da RAM que nos informam de processos, ficheiros executáveis ou ligações de rede que possam estar relacionados com *malware*, a estes também se pode chamar indicadores de compromisso (IoC).

2.2. Análise Forense Digital

Com a informatização de muitos serviços e organizações o aumento de cibercrimes tem sido constante, existindo cada vez mais ferramentas e vetores de ataque para cada sistema.

Devido a informação estar em formato digital levou a que começasse a existir novos problemas e desafios que provêm dos seguintes factos[4]:

- Dados digitais são uma representação abstrata da informação, é apenas uma sequência de *bits* e não tem nada que naturalmente prove a sua autenticidade e origem.

- Existe uma grande diversidade em dispositivos usados para criar e guardar informação digital, incluindo camaras, assistentes pessoais digitais, telemóveis e computadores. Extrair dados destes equipamentos de forma consistente é um desafio que se torna ainda mais complicado quando os dados estão escondidos, cifrados ou até mesmo fragmentados.
- Uma vez que o armazenamento digital tem um preço bastante baixo somos tipicamente confrontados com um grande volume de dados num cenário forense. Analisar estes dados enquanto se procura por alguma informação que possa ser usada como prova é uma tarefa que pode ser bastante complicada.
- Provas digitais normalmente passam por vários meios, pode ser encontrada em diversos dispositivos e em diferentes formatos.
- Dados digitais são maleáveis, e por isso existem uma grande variedade de ferramentas anti forense que podem ser usadas para mascarar ou apagar permanentemente provas digitais.
- Dados digitais podem ser voláteis e requerem uma análise atempada.

Devido a todos estes desafios são necessárias metodologias, ferramentas e alguma rapidez na análise, uma vez que se podem perder dados até mesmo na aquisição dos mesmo aquando se faz a recolha dos dados da máquina que foi atacada ou onde ocorreu o crime.

Na análise forense uma das etapas que se tem tornado cada vez mais importante é a análise de memória RAM e é nessa fase da análise forense que este projeto se foca.

2.2.1. Análise Forense na RAM

Com uma análise forense mais tradicional, em que não se faz a análise de RAM, podem ser perdidos dados importantes tais como, possíveis ligações ao exterior e/ou algum tipo de *secret key* que o *malware* ou ataque possa ter deixado.

Atualmente as técnicas de análise forense de RAM, como *Live Response* e *Memory Imaging* usadas por investigadores durante as operações de análise e apreensão envolvem análise de RAM na máquina onde foi registado o incidente ou aquisição de uma imagem da RAM para análise posterior em máquinas dedicadas. No entanto, devido à RAM ser uma memória volátil se a máquina estiver desligada, esta análise não será possível, uma vez que toda a informação foi perdida quando a máquina foi desligada.

Os sistemas operativos, tal como fazem no seu sistema de ficheiros quando um ficheiro é apagado ou deixa de ser necessário, o sistema apenas marca o ficheiro para ser eliminado, mas não o apaga imediatamente, ou seja, o ficheiro continua naquele local até que outra informação seja escrita por cima[5].

A RAM também contém regiões de armazenamento em cache que incluem diversas informações sobre o estado do sistema, o que pode ser importante para tentar perceber o que realmente aconteceu com aquele ataque e em que estado estava o sistema.

A análise de RAM tem bastantes vantagens em relação à análise da memória não-volátil. Enquanto que a memória não-volátil é utilizada para armazenar informação sem contexto de execução, a RAM permite consultar o estado e as configurações que estavam a ser utilizadas no sistema operativo e aplicações no momento da aquisição. Com esta informação, será possível cruzar os dados na memória não-volátil com a informação na RAM para encontrar incoerências entre a informação presente em ambas as memórias.

Dada a natureza dos *rootkits*, um tipo de *malware* em que o objetivo é esconder ao máximo a sua presença e atividades no sistema, a informação que possa ser recolhida na RAM poderá ajudar a encontrar informação ofuscada na memória não-volátil.

Através de uma análise de RAM temos hipóteses de recolher informação sobre o *malware* que de outra forma estaria cifrada, podendo também conseguir recolher chaves de acesso que remetem para servidores a que esse *malware* está associado.

Podemos ver que tipo de processos foram executados pelo *malware* e criar um *template* de análise que pode ser usado posteriormente para ser mais fácil identificar, ou até mesmo bloquear a execução daquele tipo de *malware*.

Contudo, existem alguns problemas relacionados com a confiança neste tipo de análise uma vez que para efetuar a recolha de uma cópia, existe a necessidade de executar algum código ou *software* na máquina que pretendemos analisar, o que pode por vezes alterar alguma da informação que era importante para análise e para recolha de provas [6].

2.3. Yara Rules

Yara é uma ferramenta *open-source* que ajuda investigadores de código malicioso a classificar amostras procurando por determinadas características. Além de analisar *malware*,

Yara pode ser usado, para analisar a natureza de ficheiros e classificar o seu conteúdo, o que é útil em análises forenses digitais[7].

O repositório Yara-Forensics contém regras para determinados tipos de ficheiros através da identificação de determinados *bytes*, podendo ser usado em combinação com soluções de antivírus *open-source* tais como ClamAV. Pode também fazer uma pesquisa num sistema usando uma regra ou combinação de várias[8].

Existem regras Yara que foram criadas e publicadas por indivíduos ou empresas de segurança para detetar software malicioso. Yara pode ser usado para construir indicadores de compromisso efetivos.

Regras Yara são como parte de uma linguagem de programação, ou seja, funcionam através da definição de diversas variáveis que contém padrões encontrados previamente numa amostra de *malware*. Se alguma das condições ou todas, dependendo da regra, forem correspondidas, a regra Yara pode então ser usada para identificar *malware* com sucesso[9].

Aquando da análise de *malware,* investigadores identificam padrões e *strings* dentro do mesmo que permitem identificar o seu grupo de ameaça, ou a família de *malware*. Ao criar uma regra Yara a partir de diversas amostras pode ser possível identificar se é do mesmo atacante, ou grupo de ataques.

Quando um analista investiga software malicioso pode criar uma regra Yara para a nova amostra que está a analisar. Esta regra pode ser usada para procurar por *malware* na sua base de dados privada ou num repositório online tal como VirusTotal [10].

Se o analista estiver numa organização que tem um *Intrusion Prevention Sytem* (IPS) ou outra plataforma usada para proteção de malware que suporta regras Yara, as mesmas podem ser usadas para resposta a incidentes para detetar software malicioso.

Organizações tais como, *National Cybersecuity and Communications Integration Center*(NCCIC) usam regras Yara para a divulgação simples e rápida de assinaturas de *malware*. Por exemplo, um ficheiro com o nome “blackenergy_v3.yara” incluído no alerta enviado, é um ficheiro que diz ao *software* onde vai ser usado, quais as sequências de *bytes* e/ou *strings* nas quais deve ser feito o alerta. Se encontrar algo que faça a correspondência vai dar o alerta ao utilizador [11].

Até agora não existiram grandes avanços em endereçar o problema da qualidade das regras Yara. Existe principalmente um artigo que aborda este problema. Neste artigo é explorada a relação entre regras Yara e *fuzzy hashing* de forma a melhorar a performance das mesmas[12]. A investigação nesta área acaba por ser muito focada em automatizar o processo das regras e criação de padrões que se traduz numa automatização da criação de IoC, de forma a acelerar o processo da análise de *malware*.

Existem ferramentas tais como o MASSE: *Modular Automated Syntactic Signature Extraction*, que funcionam com uma arquitetura cliente servidor para deteção de *malware*. Esta ferramenta incluí a criação automática de regras Yara. Este sistema é capaz de criar regras e implementá-las na sua arquitetura para posterior análise[13].

Neste documento [14], propõe-se uma ferramenta que faça análise através de instâncias em vez de padrões e dessa forma criar IoCs de forma a tornar a deteção o mais fiável possível, garantindo precisão, elasticidade e facilidade de interpretação.

No entanto, todos estes documentos e ferramentas se focam bastante na melhoria das regras Yara ou até mesmo no uso de novas ferramentas, quando já existe uma ferramenta que é bastante usada para este tipo de análise que é o Volatility.

2.4. Software

Para fazer análise forense à memória existem diversas ferramentas, e linguagens de programação usadas por essas ferramentas. Uma das ferramentas mais populares é o Volatility, e é neste que este projeto se foca principalmente. No entanto não podemos deixar de mencionar outras como IDA Pro que é também uma ferramenta bastante poderosa para análise de RAM, mas neste caso pode não ser totalmente grátis, uma vez que existem diferentes versões, algumas delas pagas. O Cuckoo *sandbox* é uma ferramenta diferente das mencionadas anteriormente, neste caso é feita uma análise dinâmica e comportamental da imagem de RAM. Como linguagem principal temos o Python, não só porque é nesta linguagem que o Volatility é desenvolvido, mas por ser compatível com os diversos sistemas operativos presentes no mercado e pela quantidade de bibliotecas externas que se podem usar. De seguida é feita uma introdução a todas estas ferramentas.

2.4.1. Volatility

Volatility é uma *framework open source* utilizada para análise forense de memória RAM e investigações digitais. Consegue analisar e extrair artefactos tanto de versões 32 ou 64-bits de sistemas operativos.

Suporta diversas versões de Linux, Windows, MacOS e até mesmo Android. Consegue analisar diversos tipos de ficheiros, sendo que o principal ficheiro utilizado é *raw memory dumps*.

A *framework* tem como intenção investigar o estado do sistema independentemente e consiste em mais de 35 plugins para análise. É possível também construir os nossos próprios plugins para esta *framework*[15].

2.4.2. IDA Pro

IDA Pro é um software que pode ser utilizado para uma análise estática. É capaz de criar mapas que mostram as instruções binárias que são realmente executadas com uma representação simbólica(*assembly*). Foram implementadas técnicas avançadas para que possa gerar código fonte em *assembly* a partir de código que foi executado pela máquina e tornar possível a sua leitura pelo utilizador [16].

2.4.3. Cuckoo Sandbox

Cuckoo é uma *sandbox* que é usada para uma análise automatizada de *malware*. Consegue analisar diversos ficheiros maliciosos tais como, executáveis, documentos *office*, ficheiros PDF ou até mesmo emails, e também websites maliciosos em diversos ambientes[17].

Consegue também seguir chamadas a *Application Programmable Interface* (API) e o comportamento do ficheiro. Esta informação é depois apresentada a alto-nível com assinaturas compreensíveis para o ser humano. Analisa também o tráfego de rede mesmo quando cifrado.

Faz análise avançada de memória do sistema virtualizado que está infetado através do Volatility e também usa regras Yara durante esta análise.

Existe a possibilidade de extrair regras Yara desta ferramenta após terminada esta análise comportamental que é feita ao *malware*.

2.5.Python

Python é uma linguagem de programação interpretada a alto nível para uso geral. O seu design tem em enfâse a leitura de código com a sua importância na indentação de código. A sua abordagem à programação orientada a objetos procura ajudar programadores a fazer código lógico e claro, seja para pequenos ou grandes projetos[18].

Python é escrita dinamicamente e *garbage-collected*, suporta diversos paradigmas de programação incluindo estrutural, particularmente procedimental, orientada a objetos e funcional.

Em vez de ter todas as funcionalidades na sua fundação, Python foi desenhado para ser altamente extensível, através de módulos. Esta modularidade compacta fê-la particularmente útil para adicionar interfaces programáveis a aplicações existentes.

Python esforça-se por ter uma sintaxe simplificada dando aos programadores escolha na sua metodologia de código. Esforça-se também para evitar otimização prematura e rejeitar *patches* a partes não críticas que podem custar um aumento marginal em velocidade e claridade.

Quando velocidade é uma característica importante é possível mover funções temporalmente críticas para módulos escritos em linguagens como C, ou usar PyPy como um compilador *just-in-time*.

2.6.Problema

Como podemos verificar anteriormente as regras Yara em conjunto com a ferramenta Volatility podem ser algo bastante útil para a análise de *malware*, no entanto o Volatility é uma ferramenta que necessita algum conhecimento de linha de comandos.

Existem também diversos repositórios de regras Yara com uma grande variedade que podem ser usadas para diversos *malwares*, no entanto o Volatility só nos permite correr um ficheiro de regras de cada vez nativamente, e é este o problema que nos propomos a resolver, ou seja, queremos que exista apenas um comando para correr diversos ficheiros de regras Yara e no final apresentar os resultados de uma forma compreensível ao utilizador. Com isto também torna a utilização deste módulo do Volatility um pouco mais simples e intuitiva para quem não está tão à vontade com o mesmo.

2.7.Síntese

Neste capítulo foram apresentadas algumas metodologias de análise forense de RAM juntamente com algumas das vantagens e desvantagens, e que cada vez mais é utilizado aquando da resposta a incidentes. Uma das grandes ferramentas utilizada para análise de *malware* na RAM é utilização de regras Yara para identificar padrões. Pode verificar-se que estas regras são usadas em diferentes ferramentas e que o seu desempenho e qualidade têm sido o principal foco de investigação. Existem diversas ferramentas que fazem a utilização destas regras e foram apresentadas algumas das que são consideradas mais importantes nesta área de análise de RAM, seja esta de uma forma estática ou dinâmica. Foi mencionado o Python como linguagem de programação pois é bastante usada devido à sua quantidade de bibliotecas com que se pode trabalhar, e facilidade de trabalhar em diversos sistemas operativos.

3. Solução Proposta

Com esta solução pretendemos resolver o problema mencionado anteriormente que consistia na limitação do módulo yarascan do Volatility não conseguir verificar múltiplos ficheiros de regras e um pouco da sua complexidade no comando que se tem de executar. Sendo que desta forma com o mesmo plugin conseguimos correr um ou vários ficheiros de regras.

Pretende-se também consolidar algumas regras das principais fontes online que desta forma podem ser descarregadas e utilizadas com este plugin, e também poderá ser realizada posteriormente a sua atualização.

3.1. Arquitetura da Solução

Na Figura 1 temos a arquitetura idealizada para esta solução, que tem por base a análise de uma imagem de memória RAM infetada através da *sandbox* Cuckoo. O Cuckoo por sua vez irá fazer uma análise dinâmica a esta imagem de RAM e para além do seu relatório e informação que esta *sandbox* gera, também cria regras Yara com base na sua análise.

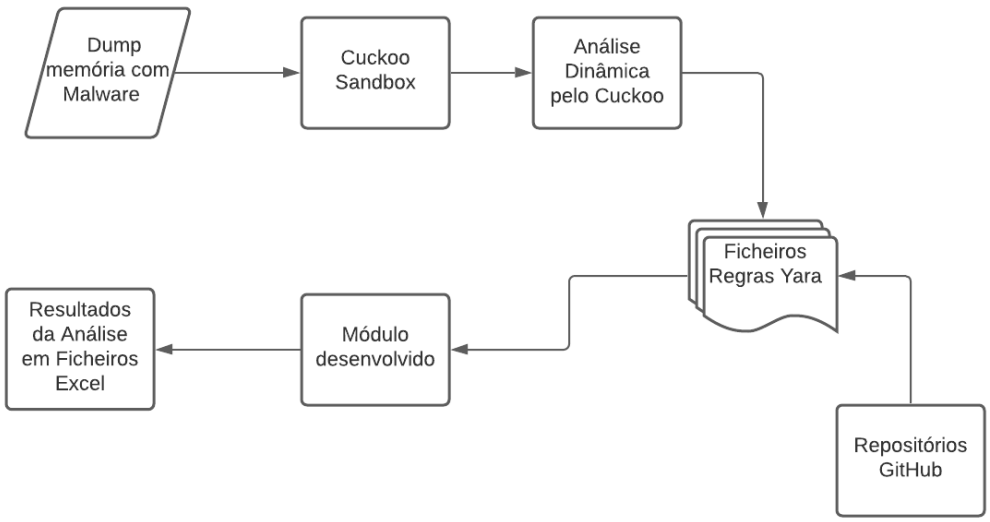


Figura 1 Arquitetura do conceito inicial

A análise dinâmica que é realizada pelo Cuckoo consiste em correr a imagem de memória RAM num ambiente virtual, normalmente uma máquina Windows 7, sendo que é realizada uma análise ao comportamento desta imagem de RAM que foi introduzida. Os possíveis

malwares que estejam nesta imagem ficam num estado de execução o que permite ao Cuckoo analisar o seu comportamento, em termos de processos, ligações de rede, ficheiros executados, chaves secretas que sejam usadas, entre outros elementos que podem ser considerados IoC, para o desenvolvimento de regras Yara. O Cuckoo apresenta-nos um relatório no final, como o apresentado na Figura 2Figura 2 Exemplo de relatório Cukoo Sandbox [19], que indica se esta imagem é maliciosa ou não e nessa imagem de relatório é onde obtemos as regras Yara.

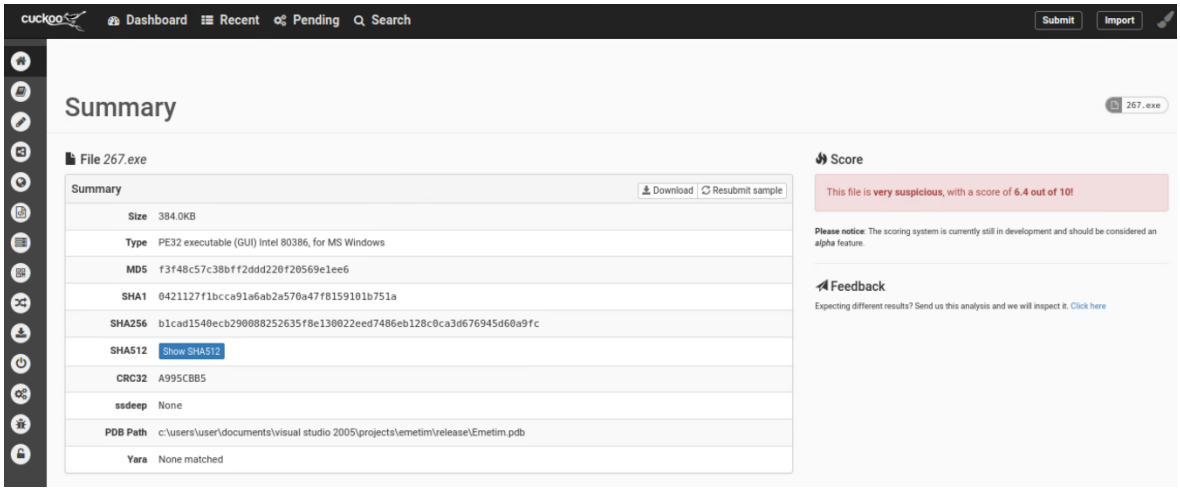


Figura 2 Exemplo de relatório Cukoo Sandbox [19]

As regras Yara criadas pelo Cuckoo serão extraídas e utilizadas pelo módulo Volatility de forma a efetuar uma análise à mesma imagem de memória RAM, mas neste caso usando regras Yara. Este módulo terá duas formas de funcionamento, pode usar regras que são criadas por nós ou pelo Cuckoo que nós introduzimos numa pasta, ou pode fazer o uso de repositórios de regras Yara. Para isto será também implementado uma forma de efetuar o *download* dos repositórios ou as suas atualizações.

No módulo desenvolvido irão existir duas opções que fazem distinção entre usar a pasta que vai conter as regras criadas pelo utilizador ou pelo Cuckoo e as regras que são recolhidas dos diversos repositórios de regras Yara, através das bibliotecas do Python que fazem esta interação com o GitHub.

Os resultados desta análise através de regras Yara são colocados em ficheiros Excel de forma que o utilizador tenha facilidade em consultar os resultados e fazer a sua exportação para outras ferramentas de forma a fazer o seu próprio relatório.

Para este módulo que se pretende desenvolver temos a necessidade de usar um módulo nativo do Volatility que é o módulo yarascan que faz a análise através de regras Yara. No entanto este módulo usualmente só funciona para um ficheiro e o objetivo deste trabalho é que funcione para vários ficheiros. Temos também como necessidade criar ficheiros Excel com os resultados dos diversos ficheiros de regras Yara usados pelo módulo.

Quando usamos o Volatility com o módulo yarascan, para além da sua limitação de usar apenas 1 ficheiro, como foi referido anteriormente, o seu *output* é normalmente apresentado no ecrã, sendo que a única forma de obtermos um ficheiro é usando técnicas de redirecionarmos o *output* da linha de comandos para um ficheiro.

Surgiu também a necessidade de utilizar repositórios com regras Yara disponíveis gratuitamente *online*, para tal existe a necessidade que o módulo possa comunicar com uma plataforma, tal como o GitHub onde muitos destes repositórios se localizam, e que possa fazer o seu *download* e eventual atualização.

3.2. Atualização dos repositórios Yara

A atualização dos repositórios Yara será feita através de uma opção no comando usado para correr o módulo. Através desta opção o módulo irá verificar se houve alterações no repositório online, atualizando os ficheiros ou fazer *download* de novos ficheiros que foram introduzidos no repositório.

Para que seja feita a atualização, primeiro tem de ser feito o *download* dos repositórios. Isto é feito através de outro comando dentro do módulo que faz este processo, sendo que coloca todas as regras de cada repositório dentro da mesma pasta.

A sua utilização é feita através de uma opção onde é introduzida uma categoria que corresponde ao nome atribuído à pasta onde foi colocado todo o repositório.

3.3. Repositórios GitHub de Regras Yara

Uma vez que as regras Yara são algo bastante utilizado a nível da cibersegurança, existe um grande desenvolvimento por parte de empresas e da comunidade neste tipo de regras que por vezes é disponibilizada através de repositórios no GitHub, por isso, de seguida será

apresentada uma breve explicação e no final uma comparação dos repositórios que se consideraram mais relevantes e que foram avaliados para serem incluídos neste módulo.

3.3.1. Yara-Rules Yara Project

O repositório Yara-Rules [20] tem um conjunto diverso de regras Yara para o qual qualquer um de nós pode contribuir. Neste repositório as regras são compiladas, classificadas e mantidas o mais atualizadas possível. Podemos verificar isto quando consultamos o seu repositório existem diversas pastas com diferentes categorias, tais como regras para cripto moedas, email, *exploit kits*, *malware* e até mesmo *malware* para dispositivos móveis.

3.3.2. ESET Malware-IoC

Este repositório é gerido pela ESET [21] que é uma empresa de renome no campo da cibersegurança, não só com os seus antivírus, mas também no ramo da investigação, por isso têm este repositório para ajudar a comunidade a combater *malware*. Estas regras Yara são criadas pelos investigadores da ESET, mas também permitem que a comunidade contribua para este repositório. Neste caso temos apenas regras Yara indicadas para *malware*, no entanto as mesmas estão separadas em diferentes famílias de *malware*, por isso se soubermos qual a família do *malware* que queremos encontrar podemos usar as regras dessa família.

3.3.3. ReversingLabs Yara Rules

Existe um repositório da ReversingLabs [22] que tem diversas regras feitas pelos analistas que se destinam a outros analistas de segurança, pessoas que dão a resposta a incidentes, *threat hunters*, ou outros que trabalhem em segurança que poderiam beneficiar da utilização de regras Yara no seu ambiente. Para uma regra estar neste repositório as regras Yara têm de ter uma precisão bastante alta sem que existam percas na qualidade e ter como objetivo não dar qualquer falso positivo. Estas regras são regularmente testadas pela ReversingLabs, e neste repositório poderá não existir uma atualização tão frequente devido a estes testes extensivos, no entanto apenas são adicionadas regras que sejam consideradas de alta qualidade após estes testes.

3.3.4. Bartblaze Yara-Rules

Neste repositório [23] existem regras Yara criadas por uma pessoa da comunidade, que contém algumas regras para diferentes tipos de *malware*, organizadas em algumas categorias que incluem ferramentas de *hacking* e *ransomware*. É atualizado com alguma frequência, e

foi referenciado numa listagem [24] onde apareciam muitos destes repositórios aqui mencionados.

3.3.5. Signature-Base

O Signature-Base é um repositório [25] criado pela comunidade como forma de ter uma base de dados das suas regras Yara e IoC que contém regras Yara de alta qualidade com o objetivo de ter o mínimo de falsos positivos, com uma estrutura clara e um formato consistente nas regras. O repositório tem uma estrutura um pouco diferente, pois neste caso o tipo de regra Yara é identificado pelo nome do seu ficheiro, sendo que tem a família de *malware* no início do seu nome.

3.3.6. McAfee Yara-Rules

Este repositório foi criado para acompanhar as investigações da McAfee Enterprise [26] o que nos permite confiar nestas regras e saber que as mesmas são precisas. A McAfee permite que possamos submeter regras de forma a melhorar o seu repositório. Estas regras estão categorizadas por pastas, das quais se pode destacar *ransomware, miners, mobile* e *malware*.

3.3.7. Yara-Forensics

Este repositório [27] não tem uma grande quantidade de ficheiros de regras Yara. As regras que contém estão divididas apenas em duas pastas, uma delas tem regras Yara relacionadas com ficheiros, ou seja, vai procurar por IoCs em ficheiros em vez de imagens de memória RAM. Para efetuar uma análise a imagens de memória RAM tem outra pasta com este tipo de regras Yara. Este repositório foi criado por alguns elementos da comunidade, no entanto é um repositório que já não tem atualizações há bastante tempo, a última atualização de regras Yara foi há cerca de 2 anos.

3.3.8. Citizen Lab malware-signatures

Este é um repositório [28] que foi criado pelo Citizen Lab que é um laboratório que pertence à Universidade de Toronto [29]. Contém algumas regras para diversas famílias de *malware*, mas não tem grande conteúdo, sendo que a quantidade de regras Yara é bastante pequena e a sua última atualização foi há 5 anos, por isso não há muito a esperar deste repositório em termos de atualizações.

3.3.9. AlienVaultLabs Malware Analysis

Este repositório [30] foi criado pela AlienVaultLabs ou mais recentemente conhecida por AT&T Cybersecurity. Contém algumas regras Yara divididas em diversas pastas que correspondem a diferentes tipos de *malware.* Neste repositório para além de regras Yara também contém alguns IoC, mas é um repositório que já se encontra bastante desatualizado, com a última atualização há 6 anos.

Tabela 2 Comparativo de repositórios GitHub com Regras Yara

	Periocidade de atualização	Comunidade ou Corporativo	Categorização das regras Yara	Principal foco das regras Yara
Yara-Rules Yara Project	Frequentemente	Comunidade do Yara Project	Divisão por família de malware	N/A
ESET Malware - IoC	Frequentemente	ESET	Divisão por família de malware	N/A
ReversingLabs Yara Rules	Frequentemente	ReversingLabs	Divisão por tipo de malware	N/A
Bartblaze Yara-Rules	Frequentemente	Comunidade de regras Yara	Divisão por tipo de malware	ransomware
Signature-Base	Frequentemente	Comunidade de regras Yara	Prefixo do nome do ficheiro	N/A
McAfee Yara-Rules	Frequentemente	McAfee Enterprise	Divisão por tipo de malware	N/A
Yara-Forensics	Raramente	Comunidade de regras Yara	Não existe	N/A
Citizen Lab Malware-Signatures	Desativado	Citizen Lab Universidade de Toronto	Ficheiro por família de malware	N/A
AlienVaultLabs	Desativado	AlienVaultLabs	Divisão por família de malware	N/A

.

Na Tabela 2 encontramos um comparativo dos diversos repositórios mencionados anteriormente. No módulo foi optado por colocar apenas repositórios que tenham uma periocidade de atualização frequente, ou seja aqueles em que as regras Yara tivessem sido atualizadas nos últimos dois anos. Foram encontrados alguns repositórios tal como o Yara-Forensics em que tinha existido algumas atualizações ao repositório há menos de 2 anos, mas não a ficheiros de regras Yara. Foram encontrados também alguns que se podem considerar desativados uma vez que não recebem atualizações há mais de 5 anos

Na altura em que se elabora este documento alguns dos repositórios utilizados foram atualizados há poucos dias, por isso podemos ver que se vão mantendo atualizados e disponibilizando novas regras ou atualizando as existentes.

Através da Tabela 2 conseguimos verificar que grande parte destes repositórios são elaborados pela comunidade, seja de uma forma mais organizada ou apenas uma pessoa ou duas que criaram um repositório com as suas regras Yara. Também vemos que quase todos tem algum tipo de categorização das regras. Esta categorização em quase todos eles é feita através de pastas de forma a facilitar a consulta do repositório e encontrarmos o tipo de regras Yara que queremos.

3.4. Síntese

Neste capítulo foi abordada a idealização da solução que se pretende implementar, foi abordado o fluxo idealizado e a utilização de diversas ferramentas, tais como o Cuckoo, Volatility, o módulo yarascan, e os repositório de regras Yara para obtermos ficheiros de regras Yara. Foi explicado como se pretende que a atualização dos repositórios de regras Yara funcione através deste módulo. Foi dada uma introdução aos repositórios GitHub que se consideraram relevantes e que foram avaliados para introdução neste módulo e foi apresentada uma tabela comparativa entre eles, sendo que destes apenas foram escolhidos para o módulo aqueles que tinham atualizações mais frequentes e o critério para esta seleção era que tinham de ter atualizações nos últimos 2 anos.

4. Desenvolvimento da Solução

Para o desenvolvimento da solução apresentada no capítulo anterior foi necessário desenvolver um módulo para o Volatility de raiz que por sua vez utiliza um módulo já existente que suporta a utilização de regras Yara.

De forma a tornar este módulo o mais simples de utilizar foram criadas diversas opções de linha de comandos que podem ser usadas para ter acesso a todas as funcionalidades criadas por este módulo.

De seguida, serão explicadas algumas das decisões tomadas e em que consiste as diversas funções deste módulo.

4.1. Bibliotecas Python

Neste módulo foi usado o Python, não só porque é a linguagem que o Volatility e os seus módulos usam, mas também porque é uma linguagem que permite trabalhar de diferentes formas, e pode ser considerada uma linguagem leve em termos computacionais.

Python tem uma grande quantidade de bibliotecas que nos ajudam a chegar ao resultado que pretendemos sem termos de criar todo o código de raiz, por vezes até encontramos diferentes bibliotecas que nos permitem aceder ao mesmo recurso, mas que têm diferentes funções e diferentes formas de serem utilizadas, havendo a necessidade de escolher qual a melhor opção para nós. Algumas destas bibliotecas já vêm incluídas com a instalação de Python, outras basta fazer a instalação e importação das mesmas.

Neste módulo uma das bibliotecas que foi usada foi o pygit2 que nos permite fazer o clone e atualização dos repositórios [31]. No entanto, existia a libgit2 que também permitia a interação com o GitHub, mas foi optado pela pygit2 pela forma simples com que era possível fazer tanto o clone e mais tarde a atualização dos repositórios [32].

4.2. Volatility Framework

O Volatility é uma das ferramentas mais populares na área de análise forense em memória RAM, funciona em qualquer plataforma uma vez que é escrita em Python que está presente

nos diversos sistemas operativos existentes. Tem um extensivo suporte para análise de diversos sistemas e/ou aplicações, desde as variadas distribuições de Linux, Windows, macOS e Android. Também consegue analisar *snapshots* de máquinas virtuais, *crash dumps* de algumas aplicações ou sistemas operativos.

O Volatility tem uma grande variedade de comandos, desde a análise de processos à análise de *Dynamic-link Library* (DLL). Para a análise de processos um dos comandos mais utilizado é o pslist que faz uma listagem de todos os processos na imagem de RAM que está a ser analisada apresentando a sua identificação, a identificação do processo pai, o ficheiro a que aquele processo corresponde e as *threads* que tem associadas a si. Na Figura 3 está um exemplo da execução deste comando, onde é possível verificar os diferentes processos nesta imagem de memória, apresentando o nome do ficheiro executável, a quantidade de *threads* para cada um, a sua identificação, entre outras informações relacionadas com os diversos processos.

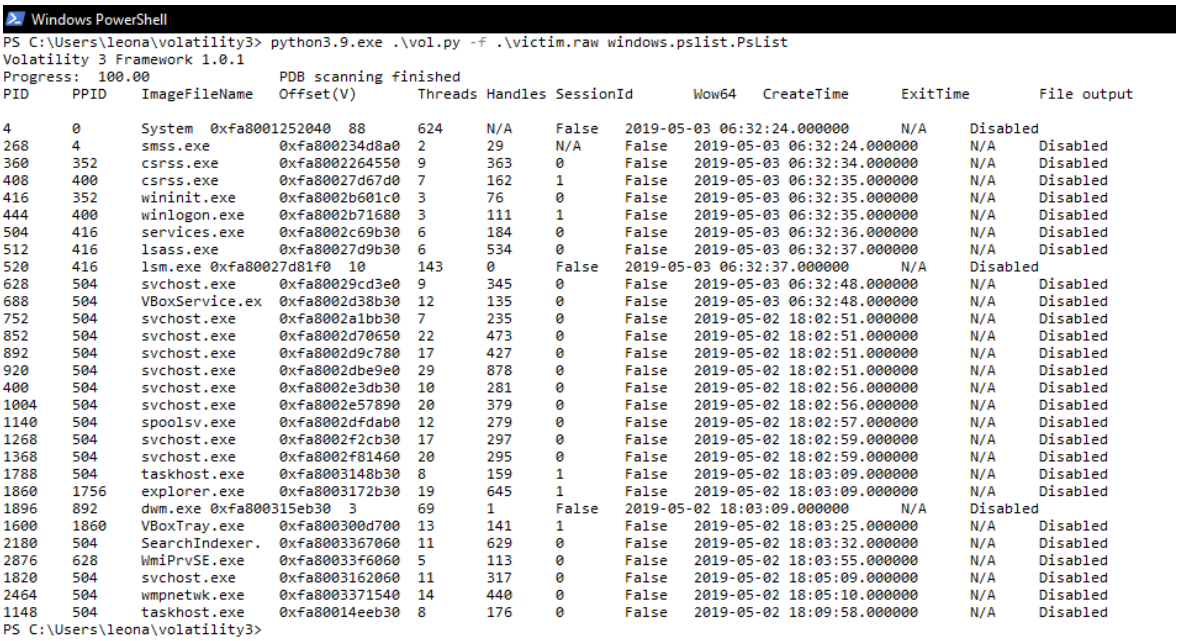


Figura 3 Exemplo de execução do comando pslist

Existe outro comando similar ao pslist que é o pstree que faz também uma listagem muito similar ao pslist no entanto adiciona uns caracteres do lado esquerdo do *output* de forma a representar os processos como se fosse uma árvore, ou seja, algo idêntico ao que o explorador de ficheiros do Windows faz com as suas pastas mostrando quais processos pertencem ao processo anterior. Na Figura 4 está um exemplo da execução deste comando onde é possível observar os asteriscos que fazem a ligação entre os diversos processos sendo que um processo sem qualquer asterisco é um processo raiz e os que têm são filhos do anterior.

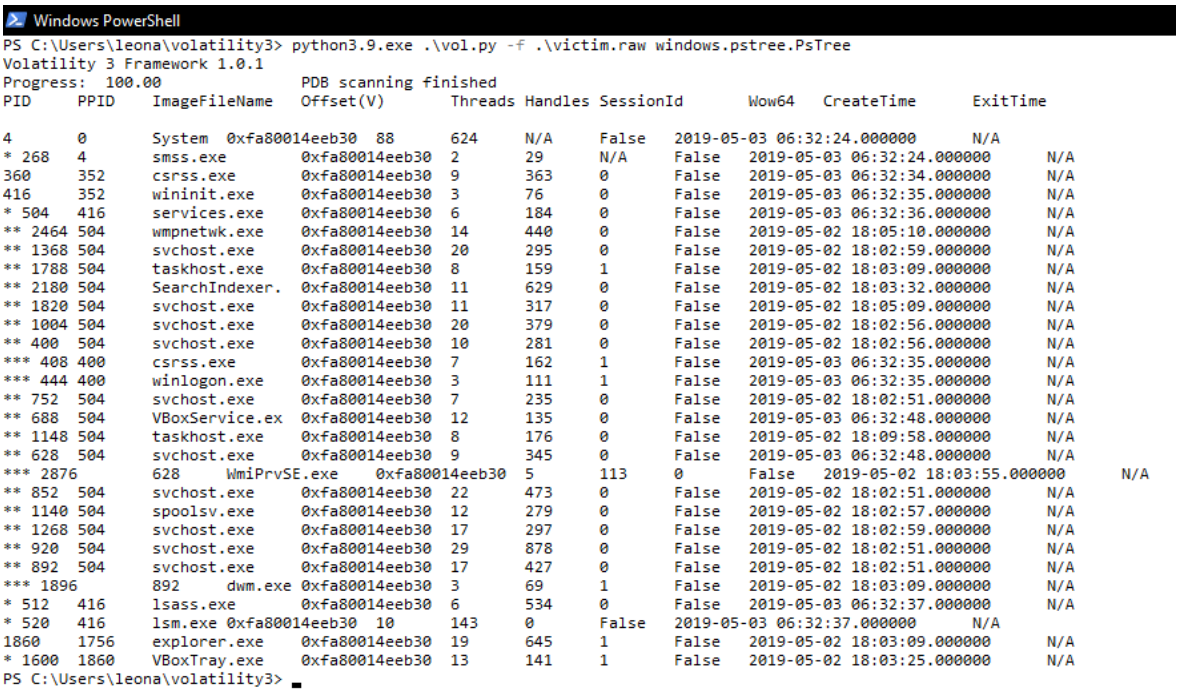


Figura 4 Exemplo de execução do pstree

O comando dlllist é um comando que faz listagem de todas as DLL presentes na imagem de memória analisada, o seu *output* pode ser algo extenso, mas é um comando importante uma vez que algum *malware* faz uso de DLL e com *output* deste comando conseguimos visualizar o processo a que cada DLL pertence, apresentando o nome do ficheiro executável e a identificação do processo, apresenta também a localização da DLL. Na Figura 5 encontra-se um exemplo de execução deste comando, onde é possível verificar as diferentes DLL presentes nesta imagem de memória RAM e qual o processo que faz uso desta DLL.

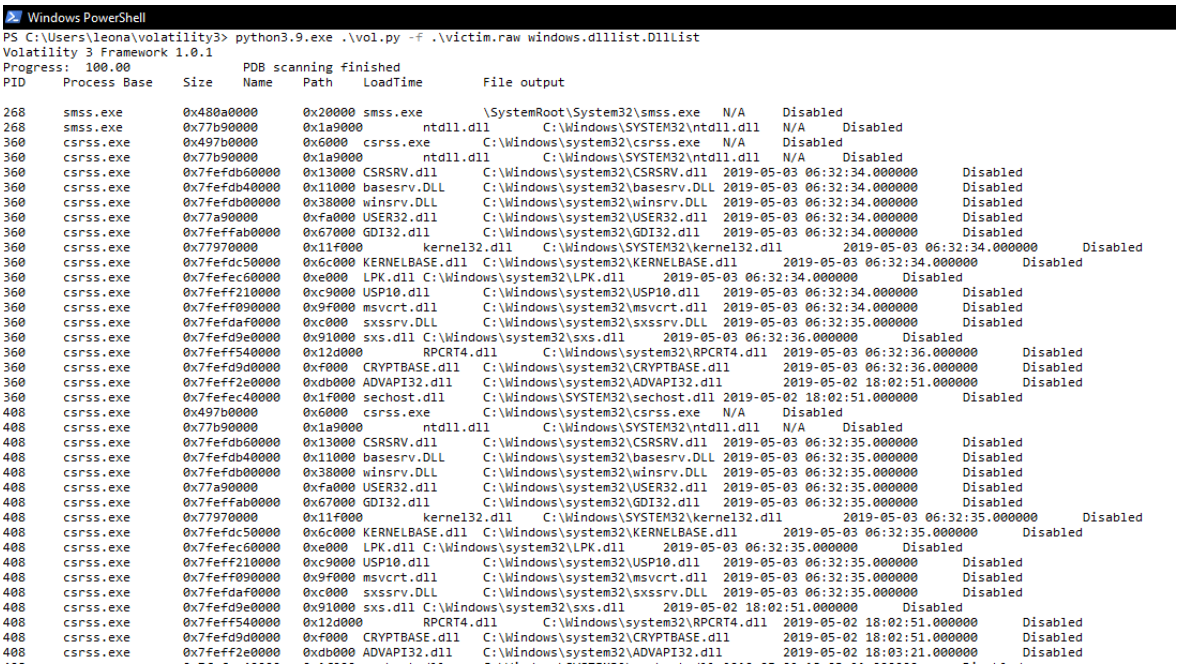


Figura 5 Execução do comando dlllist

O comando certificates permite consultar a lista de certificados digitais instalados na máquina. Por vezes pode ser útil consultar este tipo de informação uma vez que pode indicar a existência de algum tipo de *malware* na máquina. Um exemplo de importância disto foi o caso com computadores Lenovo que gerou uma grande controvérsia, pois existia um certificado digital instalado nas máquinas que tornava possível efetuar ataques de *man-in-the-middle*, ou seja, intercetar comunicações daquelas máquinas. Neste caso eram comunicações *web* seguras que se podia consultar através deste tipo de ataque [33]. Na Figura 6 podemos ver o exemplo da execução do comando certificates, onde é possível ver o resultado como uma tabela que inclui a identificação do certificado incluindo o seu nome e identificador e a que secção pertence este certificado digital.

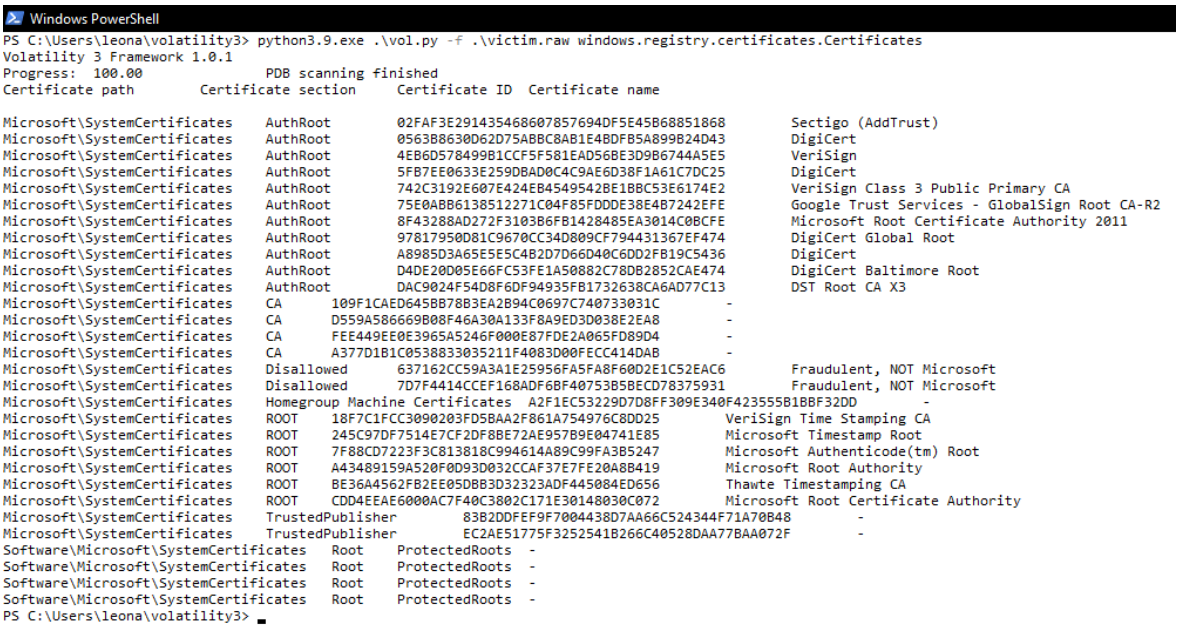


Figura 6 Exemplo de execução do comando certificates

Um comando do Volatility bastante usado quando se faz uma análise forense de RAM é o hivelist. Este comando faz a listagem das *hives* do registo do Windows presentes naquela imagem de memória RAM. Destas *hives* conseguimos tirar bastantes informações, desde nomes de utilizadores, *software* que esteja instalado naquela máquina, a zona temporal do sistema, tudo isto é possível porque o Windows guarda uma grande quantidade desta informação em chaves de registo que são organizadas em *hives*. Na Figura 7 está um exemplo de execução do comando hivelist onde são apresentadas as diferentes *hives* presentes nesta imagem de memória RAM e o caminho do ficheiro destas *hives*.

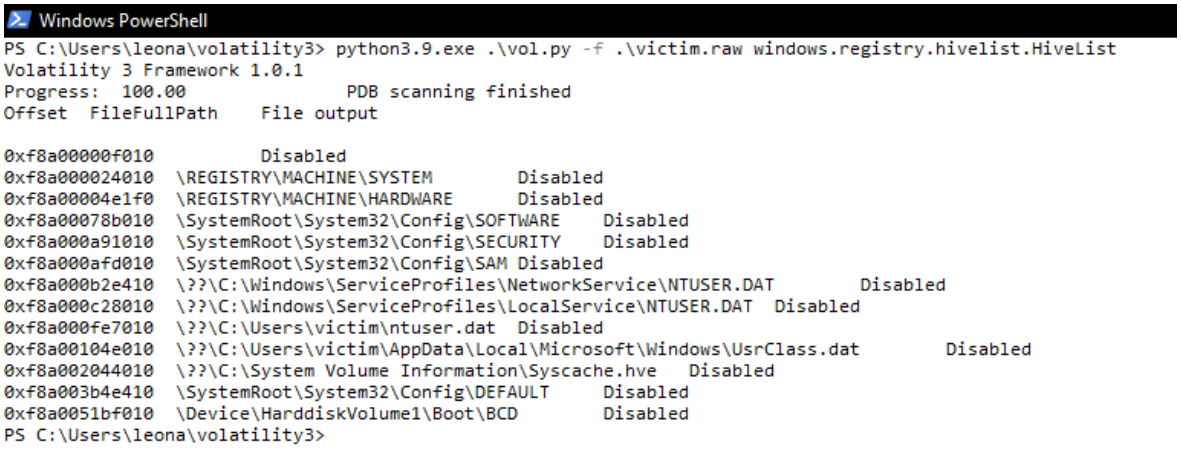


Figura 7 Exemplo de execução do comando HiveList

O Volatility tem um comando bastante simples que nos apresenta alguma informação essencial sobre o sistema que está naquela imagem de memória RAM que é o comando info. Este comando apresenta o tipo de sistema que está naquela imagem de RAM, ou seja, se é Windows, Linux, macOS e qual é a sua versão. Também nos dá alguma informação sobre a data e hora do sistema. Na Figura 8 podemos ver um exemplo da execução deste comando, onde é possível observar diversas informações sobre a imagem de memória analisada tal como o tipo de sistema operativo, a hora, quantidade de processadores entre outras informações sobre este sistema operativo.

Figura 8 Exemplo de execução do comando Info

No Volatility temos o comando yarascan que faz pesquisas na imagem de memória RAM através de uma *string* ou valor introduzido pelo utilizador, ou então através de um ficheiro que contém diversos critérios. Quando usamos um ficheiro são definidos diferentes critérios e se um ficheiro, processo ou ligação de rede corresponde ao critério definido, vai ser apresentado no resultado final.

Na Figura 9 é apresentado o resultado da execução deste comando usando um ficheiro de regras Yara, podemos observar neste output que o resultado é apresentado num formato de tabela onde contém o endereço de memória em que aquela informação foi encontrada, o nome da regra Yara que fez correspondência, o componente da regra Yara, ou seja a *string*, e por fim é apresentado o valor em hexadecimal.

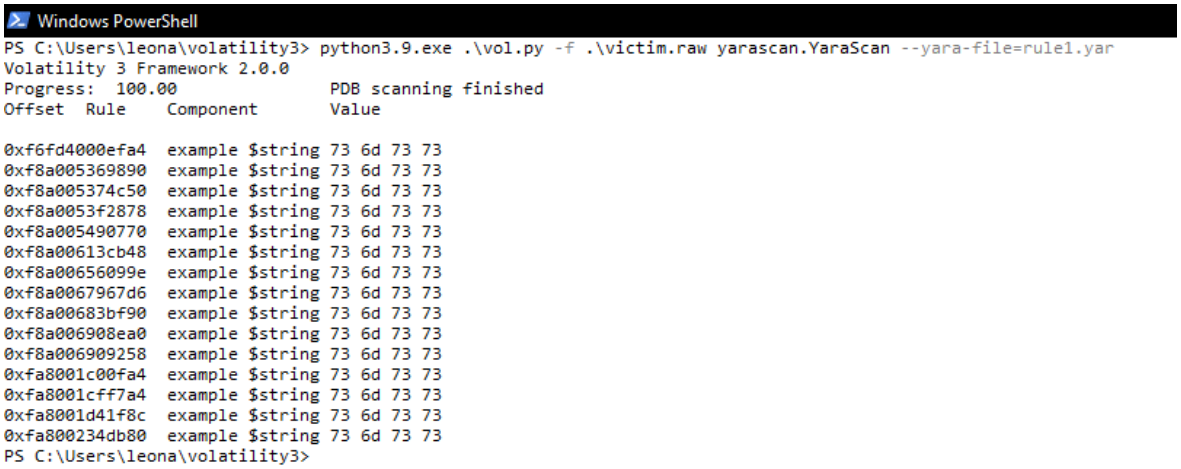


Figura 9 Resultado execução do comando yarascan

4.3.Requisitos para criar um módulo para o Volatility

Para criar um módulo para o Volatility existem alguns requisitos que têm de ser incluídos no código que estamos a desenvolver para que seja reconhecido pelo Volatility e consigamos fazer a sua utilização. Todos os módulos têm de herdar da classe *PluginInterface*. Isto é um dos passos iniciais, como podemos ver no código seguinte a classe DllList herda da classe *PluginInterface* [34].

```
1.  from volatility3.framework import interfaces
2.
3.  class DllList(interfaces.plugins.PluginInterface):
```

Após termos esta classe definida à qual podemos dar o nome que quisermos, temos de definir quais são os requisitos do nosso módulo. Existem diversos tipos de requisitos de forma que

este módulo consiga utilizar outros módulos ou receber dados a partir da linha de comandos. Podemos também incluir nestes requisitos o tipo de símbolos que o módulo vai utilizar. Tipos mais complexos como o *PluginRequirement*, que indica que o módulo faz uso de outro módulo já existente, não são pedidos ao utilizador, outros requisitos como o *ListRequirement* são pedidos ao utilizador para introduzir o valor que pretende, podemos ver no código seguinte um exemplo com diversos requisitos. Este método é executado antes do módulo em si, de forma a saber como instanciar o módulo.

```
1.  @classmethod
2.  def get_requirements(cls):
3.      return [requirements.TranslationLayerRequirement(name = 'primary',
4.                                                       description = 'Memory layer for
    the kernel',
5.                                                       architectures = ["Intel32",
    "Intel64"]),
6.              requirements.SymbolTableRequirement(name = "nt_symbols",
7.                                                  description = "Windows kernel
    symbols"),
8.              requirements.PluginRequirement(name = 'pslist',
9.                                             plugin = pslist.PsList,
10.                                            version = (1, 0, 0)),
11.             requirements.ListRequirement(name = 'pid',
12.                                          element_type = int,
13.                                          description = "Process IDs to include (all
    other processes are excluded)",
14.                                          optional = True)]
15.
```

Temos de definir um método *run* que é o método principal do módulo e que não recebe qualquer parâmetro, uma vez que todos os parâmetros são passados através do contexto, retorna uma *TreeGrid* sem quaisquer dados. Esta *TreeGrid* específica o nome de colunas e tipos que vão ser apresentados no *output*. Este método é usualmente construído com base num *generator*. No exemplo abaixo o módulo cria um filtro dos processos através do *PsList* em relação ao seu identificador. Se a lista de identificadores de processos estiver vazia apresenta todos os processos.

```
1.  def run(self):
2.
3.      filter_func = pslist.PsList.create_pid_filter(self.config.get('pid', None))
4.
5.      return renderers.TreeGrid([("PID", int),
6.                                 ("Process", str),
7.                                 ("Base", format_hints.Hex),
8.                                 ("Size", format_hints.Hex),
9.                                 ("Name", str),
10.                                ("Path", str)],
11.                               self._generator(pslist.PsList.list_processes(self.context,
12.                               self.config['primary'],
13.                               self.config['nt_symbols'],
14.                               filter_func = filter_func)))
15.
```

Podem ser introduzidos dados na *TreeGrid* sem um método *generator*, mas é uma prática comum usar um *generator* para tal. Este é o método onde usualmente se encontra a lógica principal do módulo. No código seguinte temos o exemplo deste módulo que itera por todos os processos, efetuando as diversas funções descritas neste módulo de exemplo.

```
1.  def _generator(self, procs):
2.
3.      for proc in procs:
4.
5.          for entry in proc.load_order_modules():
6.
7.              BaseDllName = FullDllName = renderers.UnreadableValue()
8.              try:
9.                  BaseDllName = entry.BaseDllName.get_string()
10.                 # We assume that if the BaseDllName points to an invalid buffer, so
11.                   will FullDllName
12.                 FullDllName = entry.FullDllName.get_string()
13.             except exceptions.InvalidAddressException:
14.                 pass
15.
16.             yield (0, (proc.UniqueProcessId,
17.                        proc.ImageFileName.cast("string", max_length =
18.                        proc.ImageFileName.vol.count, errors = 'replace'),
19.                        format_hints.Hex(entry.DllBase),
20.                        format_hints.Hex(entry.SizeOfImage), BaseDllName, FullDllName))
21.
```

Tudo isto é considerado o básico para começar o nosso módulo do Volatility. Após termos esta base podemos usar as diferentes funções e capacidades que a linguagem Python tem, e também as suas bibliotecas que nos podem ser úteis para as mais variadas coisas, tal como no caso deste projeto é necessário usar bibliotecas para a gestão dos repositórios de regras Yara ou para consultar os ficheiros de regras Yara no sistema de ficheiros da máquina em que estamos a utilizar o módulo.

4.4. Instalação do módulo

Existem alguns requisitos para efetuar a instalação deste módulo. É necessário ter o Volatility, o Python e a biblioteca Python pygit2. Para termos o Volatility basta aceder ao repositório do Volatility no GitHub e efetuar o download através do clone, para este módulo é aconselhado o uso do Volatility 3 que pode ser encontrado aqui [35]. Para o Python é aconselhado uma versão do Python 3 instalada no sistema, neste caso a instalação é diferente entre sistemas, mas desde que tenhamos o Python instalado e acessível para o Volatility, as diferentes versões do Python podem ser encontradas aqui [36]. No caso do pygit2 a instalação depende de termos o *Preferred Installer Program*(pip) instalado para a respetiva versão do Python. A instalação pode ser um pouco extensa, mas existe um guia [37]

detalhado na documentação desta biblioteca e apresenta as diferentes formas de instalação para os sistemas operativos mais comuns, Linux, Windows e macOS.

4.5. Arquitetura Solução Final

Na Figura 10 é possível ver a arquitetura final desta solução. Neste módulo existe um comando que recebe a imagem de memória e faz análise com os diversos ficheiros de regras Yara e que no final apresenta os resultados em ficheiros Excel.

Existem também comandos para fazer a gestão dos repositórios de regras Yara, isto para efetuar o download e posterior atualização das regras destes repositórios de forma a serem utilizadas pelo módulo.

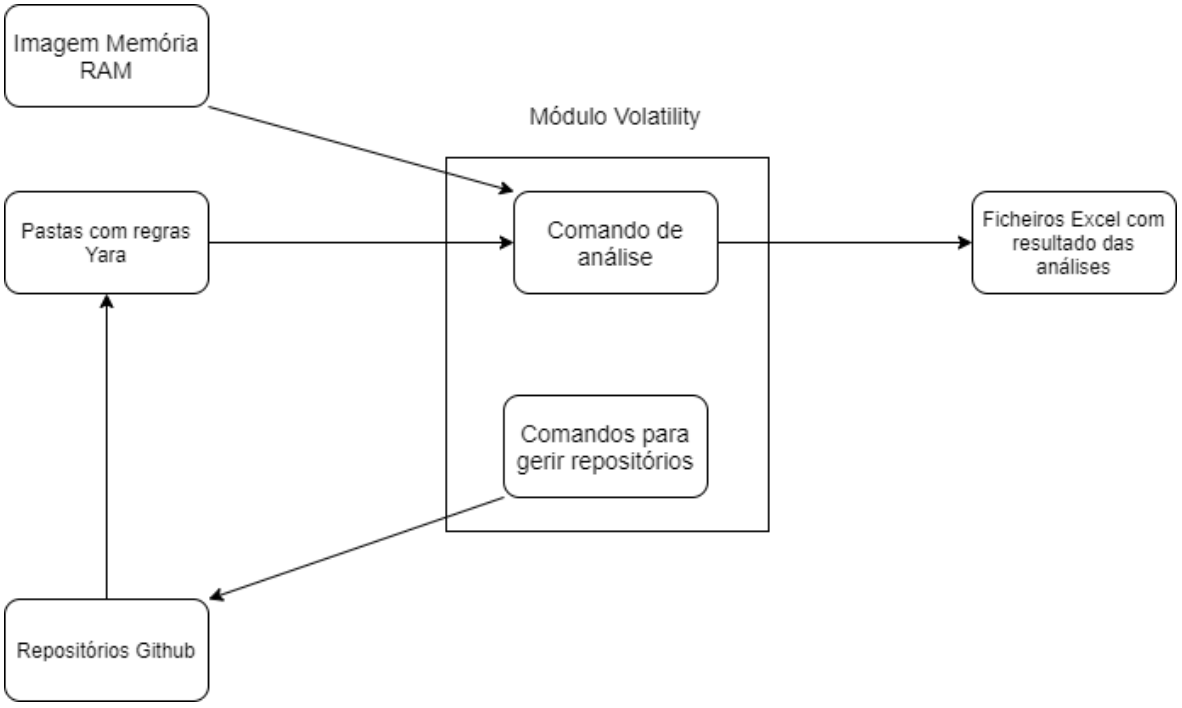


Figura 10 Arquitetura Final da Solução

Este módulo tem duas opções bastante semelhantes que é a rules e a category. A diferença entre ambas é o local onde leem os ficheiros de regras Yara. A rules vai buscar os ficheiros de regras que estão numa pasta que, pode ser definida pelo utilizador, e utiliza todos os ficheiros dessa pasta. A opção category vai buscar os ficheiros de regras aos diferentes repositórios e, quando o utilizador introduz uma categoria ao correr o comando essa categoria vai corresponder ao nome da pasta onde se encontra o repositório de acordo com a lista que se encontra neste módulo. Esta lista pode também ser alterada posteriormente de forma a personalizar os repositórios que são utilizados.

Na Figura 11 é apresentado um exemplo de como é executada a opção rules. É fornecido o número de ficheiros de regras Yara que temos para análise e esta opção vai ler esses ficheiros e executar o módulo yarascan para cada um deles, colocando o resultado em ficheiros Excel.

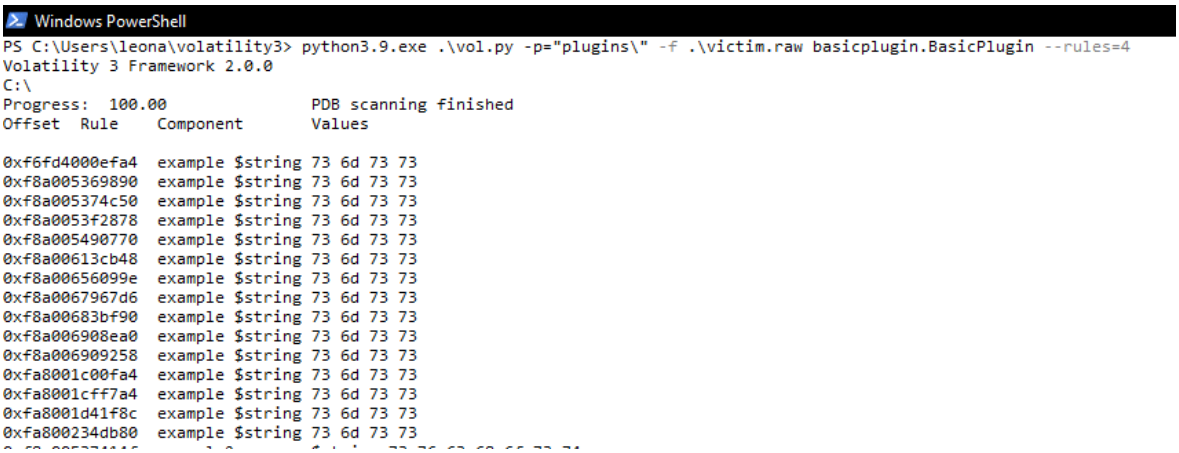


Figura 11 Exemplo execução da opção rules

Na Figura 12 é demonstrada a execução da opção category. Nesta opção é introduzido o nome do repositório que pretendemos utilizar e o módulo desenvolvido vai executar o yarascan para cada ficheiro de regras Yara que encontrar na pasta correspondente ao repositório. Tal como, na opção apresentada na Figura 11, coloca os dados em ficheiros Excel.

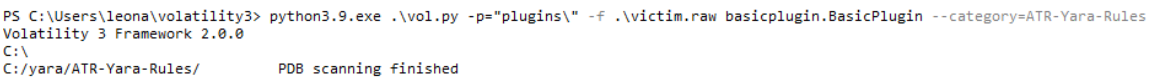


Figura 12 Exemplo de execução da opção category

Existem ainda duas opções que fazem a interação com os repositórios de GitHub. A estas opções foi dado o nome de clone e pull para serem mais fáceis de identificar uma vez que correspondem aos seus respetivos comandos de git. Na opção clone o módulo faz o *download* dos repositórios que se encontram definidos numa lista dentro do módulo e coloca-os na sua respetiva pasta. Na opção pull o módulo verifica se os repositórios já se encontram no sistema e se isto se verificar é feita então a verificação nos repositórios se existem novos ficheiros ou alterações aos ficheiros já existentes.

Na Figura 13 é apresentado o fluxo para a análise de uma imagem de RAM através do módulo criado. Neste caso é feita a leitura da imagem de RAM aquando da execução do comando para usar este módulo. O módulo verifica se a opção category está definida e se isto acontecer vai efetuar a leitura dos ficheiros de regras Yara do repositório que corresponde à categoria definida no comando. Após ter todos os ficheiros vai efetuar a análise através do módulo yarascan e por fim apresenta os resultados nos diversos ficheiros Excel.

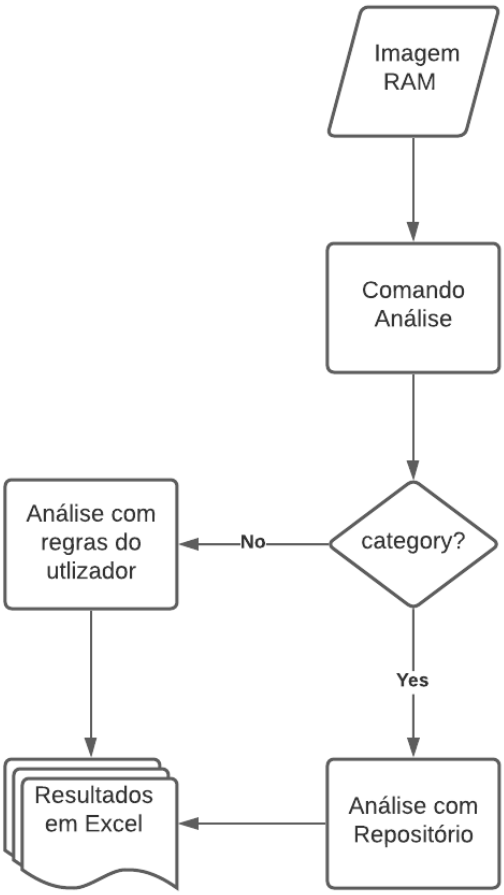


Figura 13 Fluxo da análise de RAM através do módulo

No caso de a opção category não estar definida, o módulo vai efetuar a leitura das regras que sejam criadas pelo utilizador e introduzidas na pasta necessária. Após ler esses ficheiros efetua a análise com o módulo yarascan e novamente apresenta os resultados em ficheiros Excel.

Na Figura 14 encontramos os fluxos para os dois comandos git disponibilizados juntamente com este módulo. No caso do comando clone o módulo vai verificar se o *download* dos repositórios já foi feito anteriormente. Se tiver sido efetuado, o módulo informa o utilizador que deve usar o comando pull, se os repositórios não existirem então o módulo cria toda a estrutura necessária para guardar os dados dos repositórios e faz o *download* dos mesmos.

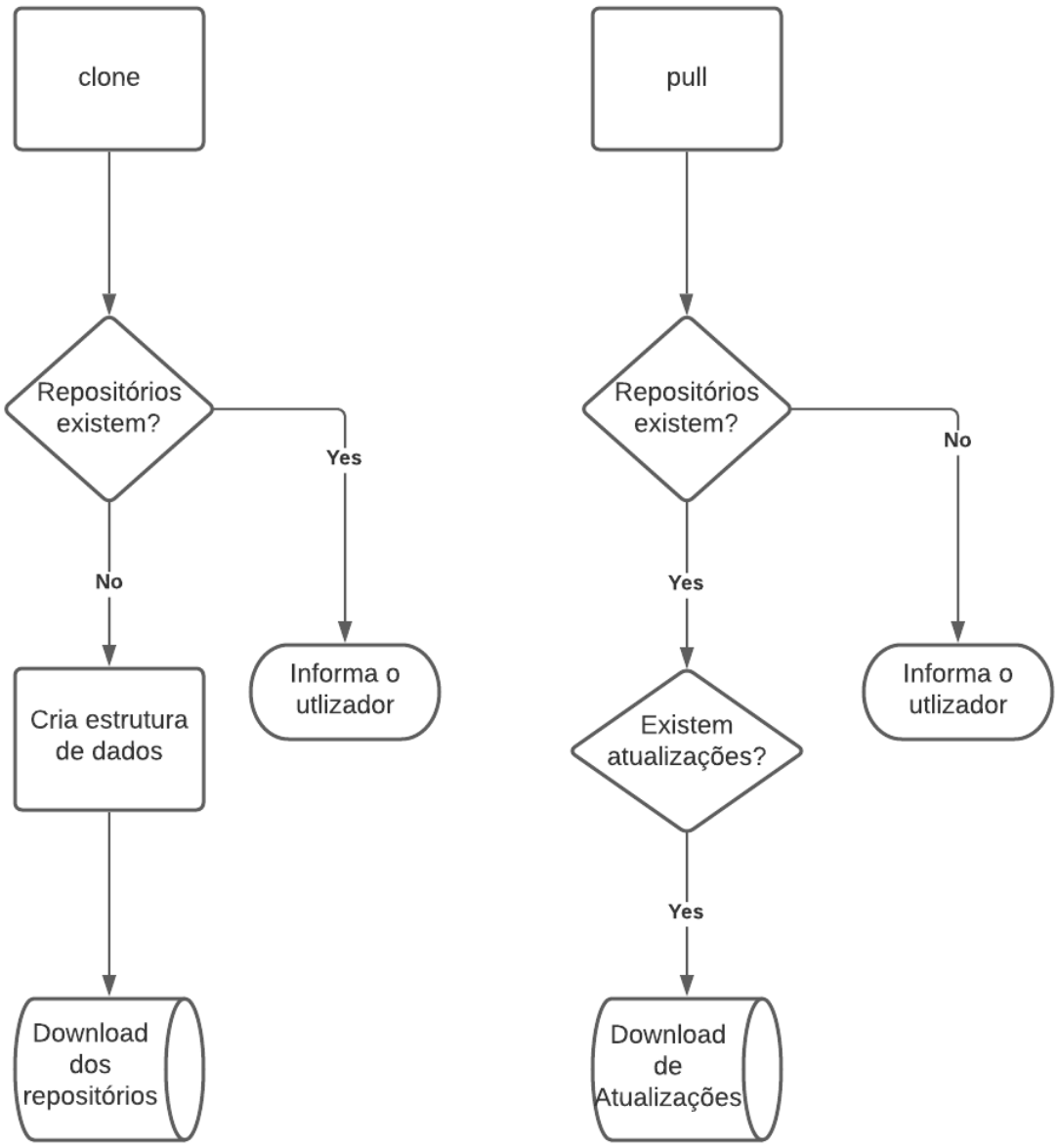


Figura 14 Fluxos dos comandos do GitHub

Se executar o comando pull em vez de clone o módulo volta a verificar se os repositórios existem. Se os repositórios não existirem informa o utilizador desta inexistência e que deve usar o comando clone. Se os repositórios já existirem no sistema, a função pull da biblioteca pygit2 verifica se existem atualizações e, se existirem faz o *download* destas atualizações.

No módulo criado existem as seguintes opções de linha de comandos:

- rules e category – Estas opções usam ambas a mesma função dentro do código. Nesta função é verificado se a category está definida, e se isto acontecer vai criar um caminho com essa categoria que corresponde à pasta do repositório e percorre essa pasta e subpastas para ir buscar os ficheiros de regras Yara dentro dessa pasta. A opção rules faz algo similar, mas neste caso recebe um valor que indica quantas regras estão na pasta definida para regras que sejam do utilizador e vai recolher esses ficheiros. Ambas as funções introduzem os ficheiros para um array que depois será utilizado pela função *_generator* de forma a utilizar o comando yarascan. No código seguinte está a representação do código destas opções e do *_generator*.

```
1.     # Function brought from the actual yarascan module of volatitlity since there was
    no way to use it without it being within the module.
2.    def _generator(self):
3.      i=0
4.      # Cycle that runs through all the files and executes the plugin yarascan, and
    writes the output into the different CSV files
5.      while i<len(self.files):
6.        self.config['yara_file'] = self.files[i]
7.        rules = yarascan.YaraScan.process_yara_options(dict(self.config))
8.        f = open('results_'+self.files[i].replace('/','_').replace(':','_')+'.csv','w')
9.        f.write(self.files[i]+'\n')
10.       i+=1
11.       f.write('Offset,Rule,Name,Value\n')
12.       layer = self.context.layers[self.config['primary']]
13.       for offset, rule_name, name, value in layer.scan(context = self.context, scanner
14.                                               = yarascan.YaraScanner(rules = rules)):
15.
16.           f.write(format(hex((format_hints.Hex(offset))))+','+
17.                   str(rule_name)+','+str(name)+','+str(value)[1:]+'\n')
18.
19.
20.
21. # Function getFiles gets all the files from the folder and puts them all into an array
    to be used later by the Yara module
22.   def getFiles(self):
23.     if self.config.get('category',None):
24.       basepath='C:/yara/'+self.config['category']+'/'
25.       print(basepath)
26.       self.folder = os.listdir(basepath)
27.       for path, subdirs, files in os.walk(basepath):
28.         for name in files:
29.           if name.endswith(".yar"):
30.             file = "file:"+os.path.join(path,name)
31.             self.files.append(file)
32.     else:
33.       folder = self.custom_rules
34.       for path,subdirs, files in os.walk(folder):
35.         for name in files:
36.           if name.endswith(".yar"):
37.             rule = "file:"+os.path.join(path,name)
38.             self.files.append(rule)
39.
```

- clone – Esta opção percorre a listagem de repositórios que se encontram no módulo e faz o *download* dos seus ficheiros para a pasta correspondente. Esta função faz uso da biblioteca pygit2 que nos permite interagir com o GitHub e faz uso da função clone_repository desta biblioteca como podemos ver no código seguinte.

```
1.  # Function to verify if the repositories exist in the system and if not download
    them all into the system creating the respective folder
2.    def gitClone(self):
3.      if not os.path.exists(self.root_path+self.folders[0]):
4.        i = 0
5.        for link in self.repositoriesURL:
6.          print(link)
7.          path = os.path.join(self.root_path,self.folders[i])
8.          print('path: '+path)
9.          repo = pygit2.clone_repository(link,path)
10.
11.         i += 1
12.         print(i)
13.       print("Repositories successfully cloned")
14.
15.     else:
16.       print("Repositories already exist, use the pull option instead")
17.     quit()
18.
```

- pull – Nesta opção existem duas funções. Uma delas foi criada para percorrer todas as pastas onde se encontram os repositórios e em cada pasta efetuar o comando pull de forma a atualizar o repositório com os dados mais recentes. Para isto é usada uma função de exemplo da biblioteca pygit2 que faz a interação com o repositório. Abaixo podemos ver o código de ambas as funções.

```
1.  # Function that checks if the repositories exist in the system and if they do and
    have updates downloads all those updates for every repository
2.    def gitPull(self):
3.      if os.path.exists(self.root_path+self.folders[0]):
4.        paths = []
5.        for path in os.listdir(self.root_path):
6.          full_path = os.path.join(self.root_path,path)
7.          paths.append(full_path)
8.
9.          for folder in paths:
10.           repo = pygit2.Repository(folder)
11.           self.pull(repo)
12.       print("Repositories successfully updated")
13.
14.     else:
15.       print("Repositories need to be cloned before using the pull option")
16.     quit()
17.
18. # Function that uses the pull method of git and is ran for each repository that
    exists
19.   def pull(self,repo, remote_name='origin'):
20.     for remote in repo.remotes:
21.         if remote.name == remote_name:
22.             remote.fetch()
23.             remote_master_id =
24.                     repo.lookup_reference('refs/remotes/origin/master').target
25.             merge_result, _ = repo.merge_analysis(remote_master_id)
26.             # Up to date, do nothing
27.             if merge_result & pygit2.GIT_MERGE_ANALYSIS_UP_TO_DATE:
28.                 return
29.             # We can just fastforward
30.             elif merge_result & pygit2.GIT_MERGE_ANALYSIS_FASTFORWARD:
31.                 repo.checkout_tree(repo.get(remote_master_id))
32.                 master_ref = repo.lookup_reference('refs/heads/master')
33.                 master_ref.set_target(remote_master_id)
34.                 repo.head.set_target(remote_master_id)
35.             elif merge_result & pygit2.GIT_MERGE_ANALYSIS_NORMAL:
36.                 repo.merge(remote_master_id)
37.                 print (repo.index.conflicts)
38.
39.                 assert repo.index.conflicts is None, 'Conflicts, ahhhh!'
40.                 user = repo.default_signature
41.                 tree = repo.index.write_tree()
42.                 commit = repo.create_commit('HEAD',
43.                                             user,
44.                                             user,
45.                                             'Merge!',
46.                                             tree,
47.                                             [repo.head.target, remote_master_id])
48.                 repo.state_cleanup()
49.             else:
50.                 raise AssertionError('Unknown merge analysis result')
51.
```

Nas opções rules e category, o módulo faz uso do módulo yarascan que já existia no Volatility. Nestas funções o que o módulo faz é procurar numa pasta por ficheiros de regras Yara e de seguida fazer a execução do yarascan com cada um deles sendo que os resultados são enviados para um ficheiro Excel em vez de serem apresentados na linha de comandos uma vez que iria existir uma grande quantidade de texto e não seria possível usar de uma forma prática.

Nas opções que efetuam a gestão dos repositórios fazemos uso da biblioteca pygit2 que tem diversas funções para fazer a comunicação com o GitHub de forma a conseguir ter estes repositórios presentes no sistema para serem utilizados com os outros comandos do módulo.

4.6. Cuckoo Sandbox

Na arquitetura da solução desenvolvida podemos ver que a *sandbox* Cuckoo não está presente como foi idealizado inicialmente, isto porque, existiram diversos problemas com esta *sandbox,* começando logo pela sua instalação que era algo complexa e causava bastantes erros. Após uma instalação bem-sucedida foram feitos diversos testes com a imagem de memória RAM que estava infetada com *malware*, no entanto o Cuckoo embora nos desse um resultado que mostrava que poderia haver *malware* naquela imagem de memória RAM nunca nos apresentava regras Yara, por isso não era possível usar regras que resultassem desta análise dinâmica e foi decidido implementar a utilização apenas com repositórios de regras Yara já existentes.

Algumas das limitações e/ou problemas encontrados com o Cuckoo foram:

- Esta *sandbox* não funciona com versões do Windows recentes para *guest*, neste caso o mais recente que funciona é o Windows 7 SP3, portanto seria impossível analisar imagens de memória RAM do Windows 10 o que se torna numa grande limitação da ferramenta.
- Foi identificado que se as versões entre a imagem de memória RAM e a máquina *guest* do Cuckoo forem diferentes não apresenta resultados fidedignos.
- Numa primeira abordagem foi feita a tentativa de instalar o Cuckoo numa máquina Windows, no entanto tinha uma forma de instalação bastante complexa

devido a ser muito dependente do Linux, por esta razão optou-se por usar uma máquina Ubuntu 20.04 LTS para efetuar a instalação desta *sandbox*.

Mesmo ao fim de ter uma instalação bem-sucedida desta *sandbox* nunca foi possível obter resultados de interesse para este projeto, pois esta *sandbox* nunca chegou a criar regras Yara com a imagem de memória RAM a que tínhamos acesso.

Este módulo pretende tornar a utilização das regras Yara no Volatility mais simples e consolidar algumas regras de fontes relevantes de forma ao utilizador ter uma base por onde começar a sua análise de RAM. Com os resultados a serem colocados em Excel torna a sua análise mais fácil e a possibilidade de exportar esses dados mais facilmente para a criação de relatórios. Pretende também permitir que novos utilizadores de regras Yara e Volatility se sintam um pouco mais à vontade com estas ferramentas.

4.7. Síntese

Neste capítulo foi abordado todo o desenvolvimento da solução final, começando por uma breve introdução ao Python e a sua importância não só para este projeto, mas também em geral, foi feita uma análise mais aprofundada do Volatility apresentando alguns dos seus módulos mais usados juntamente com o seu output. Foram apresentados os requisitos essenciais para começar a desenvolver o nosso módulo para o Volatility, explicando as diversas funções que são necessárias a qualquer módulo. Foi dada uma explicação para instalação do módulo criado e quais as bibliotecas necessárias para que este funcione em pleno. Foram apresentados os diversos fluxos que podem acontecer neste módulo através das suas opções que foram desenvolvidas dando uma descrição detalhada de cada uma e apresentando o seu código. No final foram apresentadas algumas dificuldades encontradas quando se tentou implementar o que se tinha idealizado no Capítulo 3.

5. Testes e resultados

Após o desenvolvimento efetuado no capítulo anterior existiu a necessidade de realizar testes à solução desenvolvida.

Neste capítulo são apresentados os diferentes testes de forma a demonstrar que o módulo desenvolvido atingiu os objetivos que eram pretendidos. Foram realizados testes em 2 sistemas operativos diferentes garantido o funcionamento do módulo em diferentes plataformas.

5.1. Testes de eficiência

Foram efetuados alguns testes tendo como métrica o tempo que uma pessoa levaria a correr um conjunto de ficheiros pelo método tradicional comparado com o uso deste módulo, ou seja, por cada ficheiro realizar uma execução do comando Volatility enviando o seu *output* para um ficheiro de texto. Foi usada exatamente a mesma imagem de memória e os mesmo ficheiros de regras de forma a ser uma comparação real entre o método como é feita a execução, uma vez que diferentes regras Yara e diferentes imagens de memória RAM podem traduzir em maior tempo de execução.

Da forma tradicional, ao usar o yarascan com o Volatility leva cerca de um minuto para usar 2 ficheiros de regras Yara e obter os seus resultados para um ficheiro, no entanto isto foram apenas 2 ficheiros. Ao escalar isto para uma maior quantidade de ficheiros o tempo vai aumentar devido a termos de repetir sempre o mesmo comando. Com o módulo desenvolvido, este mesmo processo leva cerca de 28 segundos para estes dois ficheiros, mas se quisermos correr diversos ficheiros para um tipo específico de *malware* temos uma facilidade em correr apenas um comando para os vários e ficheiros e poder observar os resultados finais em ficheiros Excel.

Na Figura 15 podemos consultar como são apresentados os resultados através de redirecionar o output do comando para um ficheiro de texto, neste caso temos um ficheiro algo pequeno, mas estes ficheiros por vezes têm imensas linhas das quais temos de retirar informação.

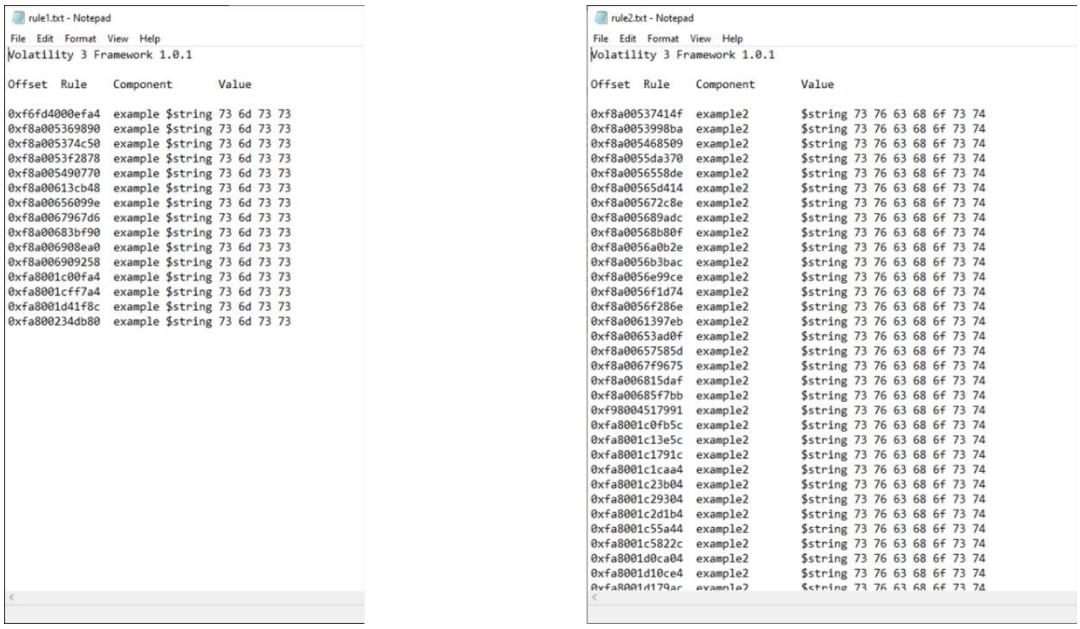
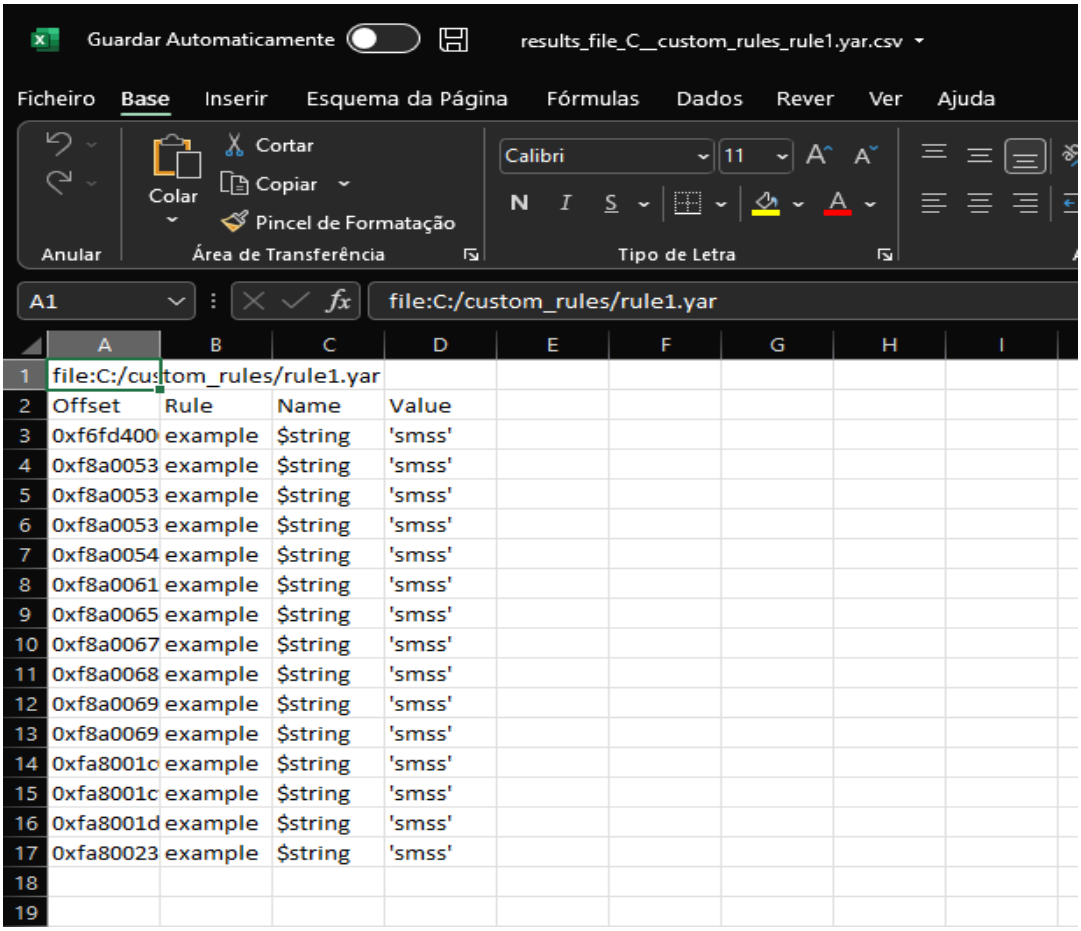


Figura 15 Resultados dos comandos pelo método tradicional

Na Figura 16 são apresentados os resultados após a execução com o módulo criado. Como podemos ver ao executar o módulo, este coloca os resultados de cada ficheiro de regras Yara no seu específico ficheiro e indicando na primeira linha e no nome do ficheiro qual foi a regra usada, no caso da Figura 16 temos o ficheiro rule1.yar, de forma a tornar a sua identificação mais simples quando usamos um repositório de regras como aqueles que estão incluídos.



	A	B	C	D
1	file:C:/custom_rules/rule1.yar			
2	Offset	Rule	Name	Value
3	0xf6fd400	example	$string	'smss'
4	0xf8a0053	example	$string	'smss'
5	0xf8a0053	example	$string	'smss'
6	0xf8a0053	example	$string	'smss'
7	0xf8a0054	example	$string	'smss'
8	0xf8a0061	example	$string	'smss'
9	0xf8a0065	example	$string	'smss'
10	0xf8a0067	example	$string	'smss'
11	0xf8a0068	example	$string	'smss'
12	0xf8a0069	example	$string	'smss'
13	0xf8a0069	example	$string	'smss'
14	0xfa8001c	example	$string	'smss'
15	0xfa8001c	example	$string	'smss'
16	0xfa8001d	example	$string	'smss'
17	0xfa80023	example	$string	'smss'
18				
19				

Figura 16 Resultados da execução do módulo em formato CSV

Tabela 3 Comparação em segundos para analisar diversas quantidades de ficheiros de regras

	Quantidade de Ficheiros Yara			
	1	2	4	8
Método tradicional	33s	60s	66s	102s
Módulo Criado	28s	28s	33s	45s

Como verificamos na Tabela 3 a diferença entre usar o módulo nativo do Volatility e usar o módulo que foi criado neste projeto não se nota quando apenas queremos usar um ficheiro de regras Yara, mas quando a quantidade de ficheiros começa a aumentar notamos uma grande diferença. Começando logo a ser possível notar esta

diferença a partir de 2 ficheiros em que verificamos que o módulo é bastante mais rápido a fazer análise em relação ao método tradicional.

Verificamos que o módulo desenvolvido é gradualmente mais rápido quando introduzimos múltiplos ficheiros de regras, com 4 ficheiros verificamos que a diferença de tempo é metade se usarmos este módulo criado, para 8 ficheiros de regras notamos que a diferença se torna ainda maior a favor do módulo criado.

O comando de clone para efetuar o *download* dos diversos repositórios de regras Yara leva cerca de 45 segundos, isto é dependente da quantidade de repositórios que temos no módulo e o tamanho dos mesmos.

O tempo que o comando pull leva vai depender da quantidade de atualizações que existem para ser feitas dos repositórios, mais uma vez vai depender do tamanho dos repositórios e da quantidade de repositórios existentes para atualizar.

Como tudo o que é relacionado com repositórios é tratado pela biblioteca pygit2, por isso no caso de algum dos repositórios que está introduzido no módulo não existir haverá um erro que será lançado por esta biblioteca e não pelo módulo.

5.2. Testes Multiplataforma

Para além destes testes de comparação entre o uso do módulo yarascan e módulo desenvolvido, foram feitos alguns testes de forma a garantir que o módulo criado funcionasse em diversos sistemas operativos sem haver uma alteração de código. Esta é uma das grandes vantagens da linguagem Python uma vez que funciona nos variados sistemas operativos que encontramos. No entanto, caminhos para ficheiros diferem entre sistemas operativos, principalmente entre Windows e Linux e por isso existiu a necessidade de verificar que todas as opções estavam a funcionar em ambos.

5.2.1. Testes sistema Windows

Para estes testes foi usada uma máquina com o Windows 10 instalada, a mesma onde foi desenvolvido todo este módulo. Foi usado um conjunto simples de regras para testar a funcionalidade do comando rules, foi também usada uma imagem de memória RAM. Tudo isto foi replicado para um sistema Linux que vai ser apresentado em mais detalhe de seguida.

Na Figura 17 está apresentado o comando clone a ser executado na máquina Windows. Podemos ver também que o comando nos apresenta o típico formato do yarascan, sendo este apresentado em formato tabela, com o endereço de memória, o nome da regra Yara com que fez correspondência e os valores. Apesar do resultado estar no ecrã foi também reencaminhado para ficheiros Excel.

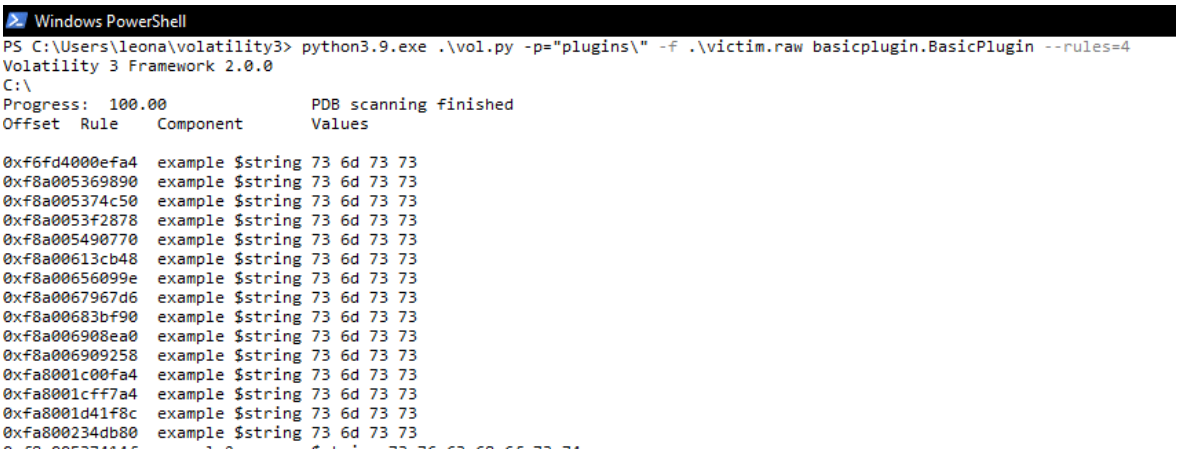


Figura 17 Execução do comando rules na máquina Windows

Na Figura 18 são apresentados os ficheiros Excel que foram produzidos pela execução do comando na Figura 17, com os resultados em ficheiros CSV por ser um formato padrão e que pode ser usado para importar os dados para outras ferramentas.

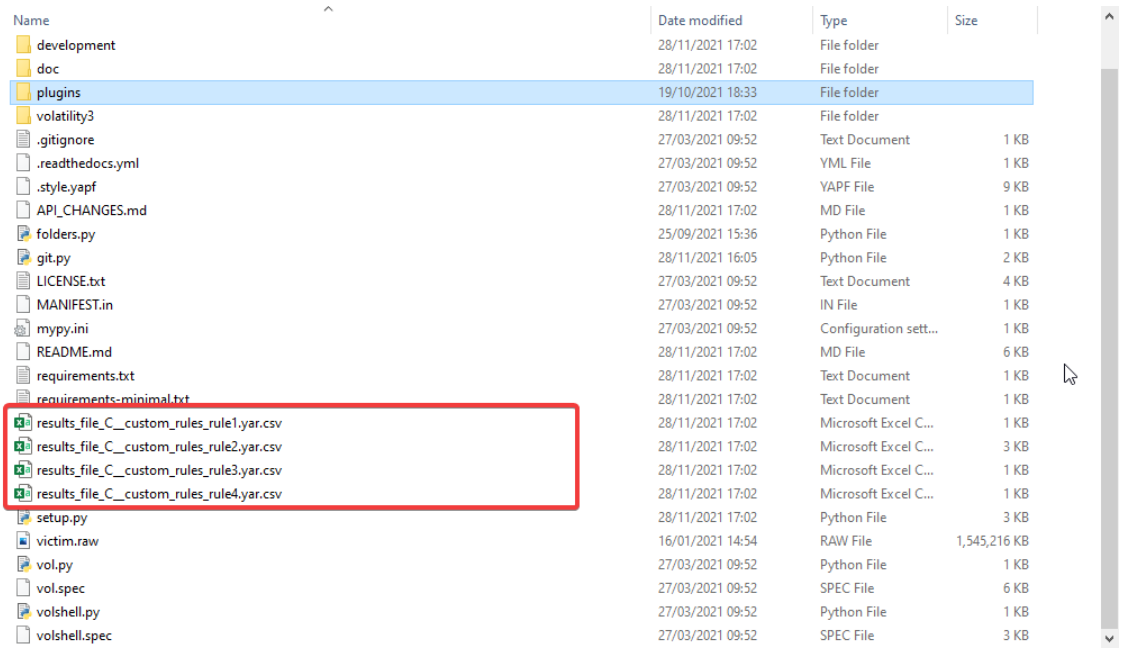


Figura 18 Ficheiros com resultados do comando rules na máquina Ubuntu

Na Figura 19 é apresentada a execução do comando clone, que faz o *download* dos repositórios introduzidos neste módulo, na máquina Windows. Como se verifica este comando percorre a lista de repositórios e coloca-os na respetiva pasta.

```
Windows PowerShell
PS C:\Users\leona\volatility3> python3.9.exe .\vol.py -p="plugins\" -f .\victim.raw basicplugin.BasicPlugin --clone
Volatility 3 Framework 2.0.0
C:\
https://github.com/Yara-Rules/rules.gitnning finished
path: C:\yara/rules/
1
https://github.com/bartblaze/Yara-rules.git
path: C:\yara/Yara-rules/
2
https://github.com/reversinglabs/reversinglabs-yara-rules.git
path: C:\yara/reversinglabs-yara-rules/
3
https://github.com/eset/malware-ioc.git
path: C:\yara/malware-ioc/
4
https://github.com/Neo23x0/signature-base.git
path: C:\yara/signature-base/
5
https://github.com/advanced-threat-research/Yara-Rules.git
path: C:\yara/ATR-Yara-Rules/
6
```

Figura 19 Execução do comando clone na máquina Windows

Na Figura 20 está apresenta a estrutura de ficheiros que foi criada pelo comando clone. Como se observa na Figura 20 temos diversas pastas que correspondem aos diferentes repositórios cujo *download* foi efetuado pelo comando clone.

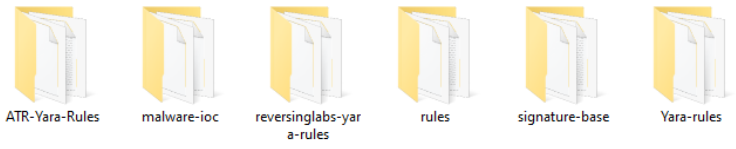


Figura 20 Estrutura de ficheiros criada pelo comando clone na máquina Windows

5.2.2. Testes sistema Linux

Para efetuar estes testes foi criada uma máquina virtual de Ubuntu 20.04 LTS onde foram instaladas apenas as dependências necessárias para executar o módulo desenvolvido, e foram usadas as mesmas regras Yara e imagem de memória RAM que foram usadas para os testes em Windows.

Na Figura 21 podemos verificar a execução do módulo desenvolvido com a sua opção rules, como se verifica o módulo funciona sem quaisquer problemas nesta máquina Linux.

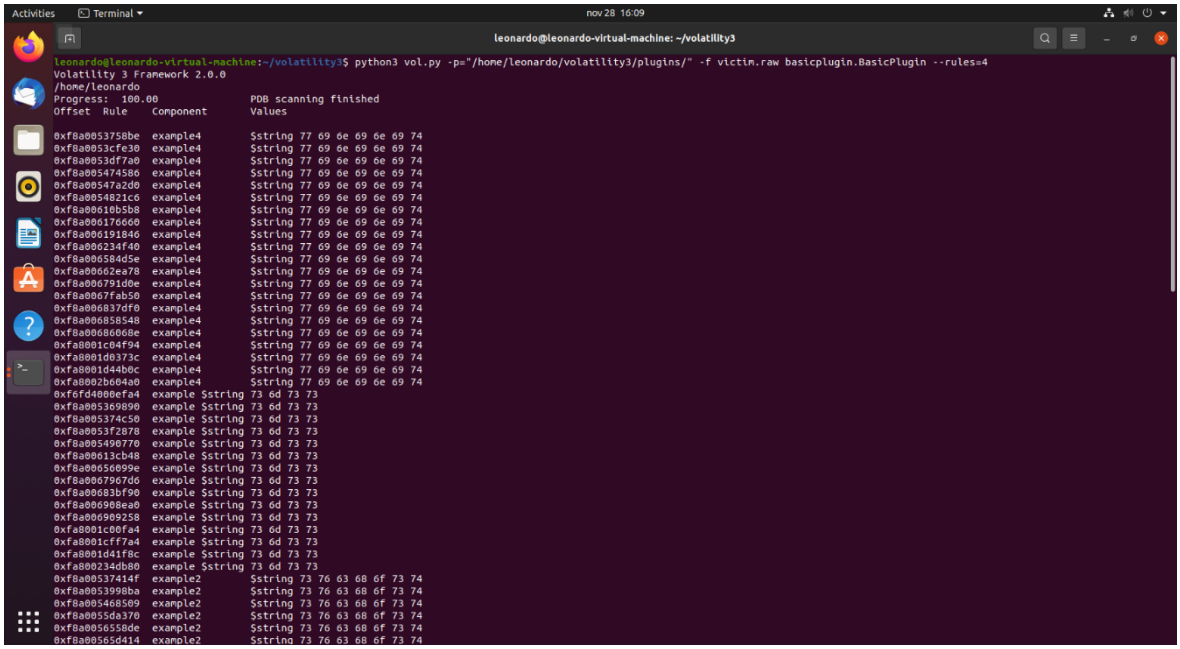


Figura 21 Execução de comando rules em Ubuntu

Na Figura 22 verificamos que tal como pretendido os ficheiros em formato CSV continuam a ser criados quando executamos o módulo desenvolvido.

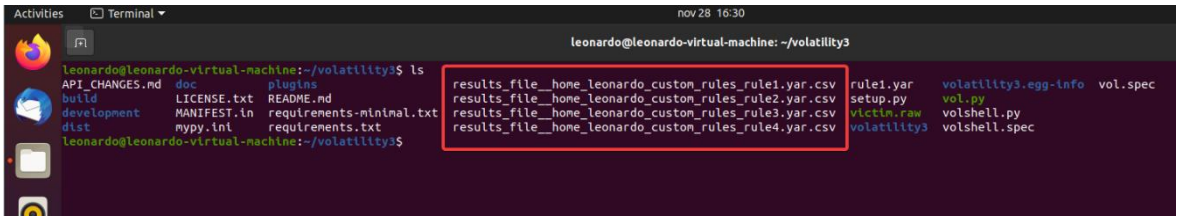


Figura 22 Ficheiros com resultados do comando rules na máquina Ubuntu

Na Figura 23 verificamos que o comando para efetuar o download dos repositórios também funciona como pretendido numa máquina Linux.

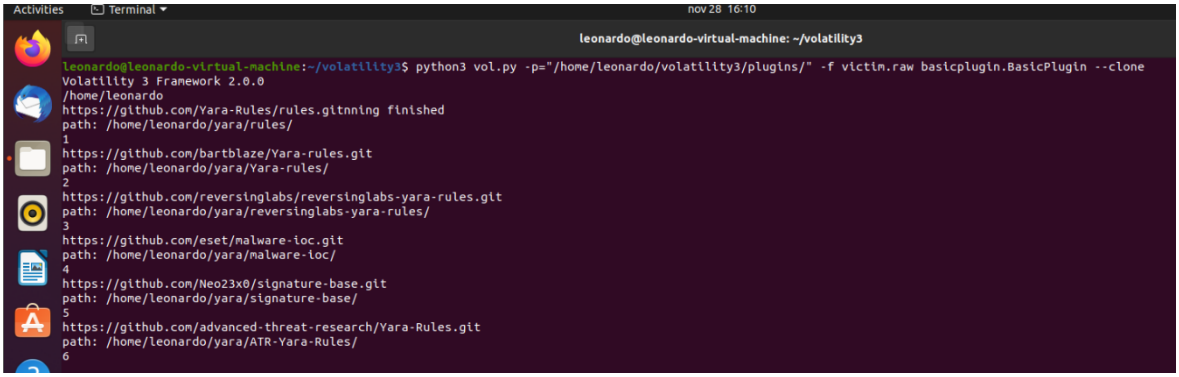


Figura 23 Execução do comando clone na máquina Ubuntu

Na Figura 24 podemos verificar que o comando criou a estrutura de ficheiros necessária para efetuar o *download* de todos os repositórios que foram incluídos no módulo desenvolvido.

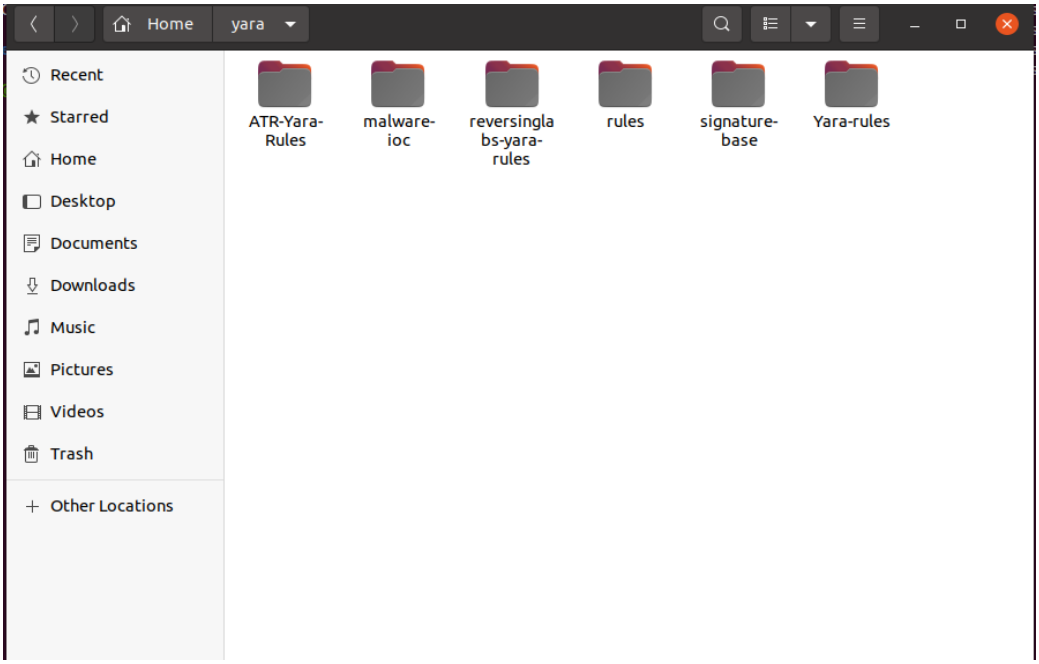


Figura 24 Estrutura de ficheiros criada pelo comando clone na máquina Ubuntu

Com estes testes foi verificado que existia uma diferença no comando para executar o módulo criado. O Volatility em si quando está em Windows não necessita de usar um caminho absoluto para indicar a pasta onde se encontram os módulos criados. No entanto no Linux é necessário dar um caminho absoluto ou o Volatility não consegue ler o módulo criado. Esta diferença pode se verificar na Figura 25 onde foi executado o mesmo comando em ambas as máquinas e foi assinalada a diferença entre eles.

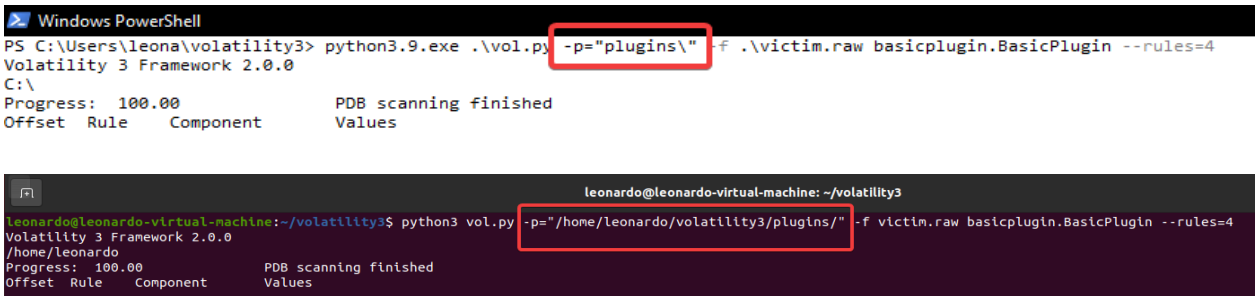


Figura 25 Comparação entre o comando em máquinas Windows e Linux

Com este projeto foi possível criar um módulo Volatility que automatiza o processo de utilização do módulo yarascan já existente, tornando também esta utilização mais eficiente e rápida caso queiramos usar múltiplos ficheiros de regras Yara de forma a obter resultados para diferentes assinaturas de *malware* que tenhamos anteriormente criado ou se quisermos podemos usar também as regras dos repositórios que se encontram neste módulo. Torna a utilização de regras Yara com o Volatility mais simples uma vez que o comando para utilizar este módulo é mais simples que usar o módulo de yarascan nativo do Volatility, especialmente no Volatility 3 em que a nomenclatura de módulos foi alterada.

Foi feita a participação no concurso *Volatility Contest* realizado pela *Volatility Foundation* que é um concurso anual onde a fundação que gere o Volatility e os seus módulos avalia os módulos submetidos pela comunidade e são avaliados por esta fundação de forma a encontrar os melhores. Este concurso não se limita a módulos, pode ser também uma aplicação que use o Volatility como uma biblioteca e faça tarefas diversas, mas uma delas tem de ser relacionada com análise forense de memória usando as APIs do Volatility.

5.3. Síntese

Neste capítulo foram apresentados os diversos resultados dos testes. Inicialmente foi efetuada uma comparação em termos de quanto tempo demorava a executar diversas quantidades de ficheiros de regras Yara entre o módulo yarascan ou utilizar o módulo desenvolvido. Foi evidente que com o escalar do número de ficheiros o módulo tinha uma grande vantagem em termos de tempo. Foram também feitos testes em diferentes sistemas operativos, neste caso Linux e Windows, e verificou-se que o módulo criado funcionava em pleno em ambos os sistemas. No entanto existe uma pequena diferença no comando para executar o módulo que provém da forma como o Volatility permite utilizar módulos criados. Por fim foram apresentados os objetivos que foram cumpridos com o desenvolvimento deste módulo, onde também é mencionada a participação num concurso para qualquer ferramenta ou módulo que utilize o Volatility.

6. Conclusão

Este projeto foi desenvolvido com o intuito de criar um módulo para o Volatility que por sua vez melhora o uso das regras Yara.

Com a crescente quantidade dispositivos presentes no nosso dia-a-dia, e o crescente número de ataques informáticos a cibersegurança e a análise forense digital tornam-se cada vez mais importantes, e com isto também a análise forense de RAM se torna importante pois contém informação que o armazenamento não-volátil não contém. Por estas razões existe cada vez mais uso de ferramentas que façam este tipo de análise sendo uma das principais o Volatility e por sua vez as regras Yara para identificação de *malware.*

Após se abordar a necessidade deste tipo de análise forense foi apresentado no Capítulo 2 as metodologias de análise forense de RAM demonstrando algumas vantagens e desvantagens da mesma. Foi dada uma explicação do que são regras Yara e foram apresentados alguns exemplos da sua utilização e da sua importância na área de cibersegurança. Foram apresentadas algumas ferramentas para além do Volatility que permitem a análise de RAM de diferentes formas, mencionou-se a linguagem de programação Python por ser bastante usada devido à sua quantidade de bibliotecas e funcionar em qualquer dos sistemas operativos que se encontram no mercado.

Com base nestas ferramentas e conceitos, no Capítulo 3 foi apresentada uma ideia inicial daquilo que se pretendia implementar e o fluxo geral que seria utilizado, tal como a utilização de outras ferramentas para além do Volatility e do seu módulo yarascan. Foi apresentado como seria feita a atualização dos diversos repositórios GitHub que se encontram presentes neste módulo, foram descritos os principais repositórios encontrados e que foram considerados para incluir neste módulo. De seguida foi criada uma tabela comparativa entre eles e apenas alguns foram selecionados sendo que o critério era que tinham de ter atualizações de regras Yara nos últimos 2 anos.

Deste conceito inicial começou a ser desenvolvida uma solução que fosse ao encontro deste conceito no Capítulo 4. Foi dada uma breve introdução à linguagem de programação Python e a sua importância para esta área, foram apresentados alguns comandos e os seus resultados do Volatility de forma a dar uma visão geral desta ferramenta e o quão importante é na análise forense de RAM. Explicou-se como se procede à instalação das dependências para

ser possível utilizar o módulo criado. Foram apresentados os diferentes fluxos de execução deste módulo juntamente com o código desenvolvido para cada função. Também foram apresentadas algumas dificuldades que se encontraram aquando da implementação do conceito inicial.

Por último foram apresentados os testes efetuados com este módulo comparando-o com o uso do módulo nativo, yarascan, do Volatility. Nestes testes podemos verificar que este módulo ao automatizar o processo do uso do yarascan torna mais rápido e eficiente a consulta de diversos ficheiros de regras Yara. Podemos também verificar que os seus resultados aparecem de uma forma mais amigável para o utilizador e permite a sua fácil exportação para outras ferramentas ou para um relatório. Foi também aqui descrito que foi feita a participação no concurso *Volatility Contest* que todos os anos analisa diversos módulos e ferramentas que façam a utilização do Volatility de alguma forma.

6.1. Trabalho Futuro

Esta solução ajuda a mitigar algumas limitações do módulo yarascan, no entanto existem algumas melhorias que poderiam ser feitas para se tornar ainda mais útil na análise forense de RAM.

Uma das principais melhorias seria tornar as categorias em famílias ou tipos de *malware*. Na sua versão atual as categorias correspondem ao repositório de onde as regras provém, mas o ideal seria que por cada repositório as regras fossem categorizadas de acordo com aquilo que verificam.

Existiu uma grande limitação inicial que foi o encontrar imagens de memória RAM com *malware* disponibilizadas. Embora tenhamos uma imagem de RAM não foi possível encontrar outras que fossem possíveis de testar com sucesso. Isto é um problema a larga escala uma vez que não existe um repositório fiável onde possamos efetuar o *download* deste tipo de ficheiros. Ideal seria criar um repositório que albergasse este tipo de ficheiros para ser consultado por pessoas que tivessem interesse em testar ferramentas que façam a análise forense de RAM.

Referências Bibliográficas

[1] "IT Insight: The seamless connection between the digital and real world." https://eu.seacoastonline.com/story/business/2021/03/25/insight-seamless-connection-between-digital-and-real-world/7006217002/ (accessed Nov. 27, 2021).

[2] "State of IoT 2021: Number of connected IoT devices growing 9% to 12.3 B." https://iot-analytics.com/number-connected-iot-devices/ (accessed Nov. 27, 2021).

[3] "DDoS attack that disrupted internet was largest of its kind in history, experts say | Hacking | The Guardian." https://www.theguardian.com/technology/2016/oct/26/ddos-attack-dyn-mirai-botnet (accessed Nov. 27, 2021).

[4] E. J. Delp, "Digital forensics [From the Guest Editors]," doi: 10.1109/MSP.2008.931089.

[5] T. Vidas, "Currently 'In Submission' to JDFP (some content may change before publication) THE ACQUISITION AND ANALYSIS OF RANDOM ACCESS MEMORY," Accessed: Nov. 01, 2021. [Online]. Available: https://users.ece.cmu.edu/~tvidas/papers/JDFP06.pdf.

[6] T. Vidas, "The Acquisition and Analysis of Random Access Memory," *J. Digit. Forensic Pract.*, vol. 1, no. 4, pp. 315–323, Jun. 2007, doi: 10.1080/15567280701418171.

[7] "Welcome to YARA's documentation! — yara 4.1.0 documentation." https://yara.readthedocs.io/en/stable/ (accessed Nov. 01, 2021).

[8] Jaramillo Luis Eduardo Suástegui, "Malware Detection and Mitigation Techniques: Lessons Learned from Mirai DDOS Attack," Jul. 16, 2018. https://www.jisem-journal.com/download/malware-detection-and-mitigation-techniques-lessons-learned-from-mirai-ddos-attack.pdf (accessed Nov. 01, 2021).

[9] "Explained: YARA rules - Malwarebytes Labs | Malwarebytes Labs." https://blog.malwarebytes.com/security-world/technology/2017/09/explained-yara-rules/ (accessed Nov. 01, 2021).

[10] "What's VT Hunting? – VirusTotal." https://support.virustotal.com/hc/en-us/articles/360000363717-What-s-VT-Hunting- (accessed Nov. 20, 2021).

[11] NCCIC, "Using Yara for Malware Detection," *National Cybersecurity and Communications Integration Center*, Jun. 2015. https://us-cert.cisa.gov/sites/default/files/FactSheets/NCCIC ICS_FactSheet_YARA_S508C.pdf (accessed Nov. 13, 2021).

[12] N. Naik, P. Jenkins, N. Savage, L. Yang, K. Naik, and J. Song, "Augmented YARA Rules Fused with Fuzzy Hashing in Ransomware Triaging," *2019 IEEE Symp. Ser. Comput. Intell. SSCI 2019*, pp. 625–632, Dec. 2019, doi: 10.1109/SSCI44817.2019.9002773.

[13] F. Biondi, F. Déchelle, A. Legay, and A. L. Masse: Modular, "Automated Syntactic Signature Extraction," pp. 1–2, 2017, Accessed: Nov. 17, 2021. [Online]. Available: http://blog.joesecurity.org/2015/02/.

[14] C. Doll, A. Sykosch, M. Ohm, and M. Meier, "Automated Pattern Inference Based on Repeatedly Observed Malware Artifacts | Proceedings of the 14th International Conference on Availability, Reliability and Security." https://dl.acm.org/doi/abs/10.1145/3339252.3340510 (accessed Nov. 17, 2021).

[15] "Volatility: The open source framework for memory forensics - Open Source For You." https://www.opensourceforu.com/2016/10/volatility/ (accessed Nov. 01, 2021).

[16] "IDA Pro – Hex Rays." https://hex-rays.com/ida-pro/ (accessed Nov. 13, 2021).

[17] "Cuckoo Sandbox - Automated Malware Analysis." https://cuckoosandbox.org/ (accessed Nov. 01, 2021).

[18] "About Python™ | Python.org." https://www.python.org/about/ (accessed Nov. 01, 2021).

[19] "Cuckoo Sandbox Overview | Varonis." https://www.varonis.com/blog/cuckoo-sandbox/ (accessed Nov. 22, 2021).

[20] "GitHub - Yara-Rules/rules: Repository of yara rules." https://github.com/Yara-Rules/rules (accessed Nov. 22, 2021).

[21] “GitHub - eset/malware-IoC: Indicators of Compromises (IoC) of our various investigations.” https://github.com/eset/malware-IoC (accessed Nov. 22, 2021).

[22] “GitHub - reversinglabs/reversinglabs-yara-rules: ReversingLabs YARA Rules.” https://github.com/reversinglabs/reversinglabs-yara-rules (accessed Nov. 22, 2021).

[23] “GitHub - bartblaze/Yara-rules: Collection of private Yara rules.” https://github.com/bartblaze/Yara-rules (accessed Nov. 22, 2021).

[24] “GitHub - InQuest/awesome-yara: A curated list of awesome YARA rules, tools, and people.” https://github.com/InQuest/awesome-yara (accessed Nov. 22, 2021).

[25] “GitHub - Neo23x0/signature-base: Signature base for my scanner tools.” https://github.com/Neo23x0/signature-base (accessed Nov. 22, 2021).

[26] “GitHub - advanced-threat-research/Yara-Rules: Repository of YARA rules made by McAfee Enterprise ATR Team.” https://github.com/advanced-threat-research/Yara-Rules (accessed Nov. 22, 2021).

[27] “Xumeiquer/yara-forensics: Set of Yara rules for finding files using magics headers.” https://github.com/Xumeiquer/yara-forensics (accessed Nov. 26, 2021).

[28] “citizenlab/malware-signatures: Yara rules for malware families seen as part of targeted threats project.” https://github.com/citizenlab/malware-signatures (accessed Nov. 26, 2021).

[29] “The Citizen Lab - University of Toronto.” https://citizenlab.ca/ (accessed Nov. 26, 2021).

[30] “AlienVaultLabs/malware_analysis at master · AlienVault-Labs/AlienVaultLabs.” https://github.com/AlienVault-Labs/AlienVaultLabs/tree/master/malware_analysis (accessed Nov. 26, 2021).

[31] “Welcome to pygit2’s documentation! — pygit2 0.26.0 documentation.” https://pygit2.readthedocs.io/en/latest/ (accessed Nov. 14, 2021).

[32] “libgit2.” https://libgit2.org/ (accessed Nov. 14, 2021).

[33] “Lenovo PCs ship with man-in-the-middle adware that breaks HTTPS connections [Updated] | Ars Technica.” https://arstechnica.com/information-

technology/2015/02/lenovo-pcs-ship-with-man-in-the-middle-adware-that-breaks-https-connections/ (accessed Nov. 25, 2021).

[34] "How to Write a Simple Plugin — Volatility 3 1.1.1 documentation." https://volatility3.readthedocs.io/en/latest/simple-plugin.html (accessed Nov. 23, 2021).

[35] "GitHub - volatilityfoundation/volatility3: Volatility 3.0 development." https://github.com/volatilityfoundation/volatility3 (accessed Nov. 24, 2021).

[36] "Download Python | Python.org." https://www.python.org/downloads/ (accessed Nov. 24, 2021).

[37] "Installation — pygit2 1.7.1 documentation." https://www.pygit2.org/install.html#quick-install (accessed Nov. 24, 2021).