MINISTÉRIO DA DEFESA
EXÉRCITO BRASILEIRO
DEPARTAMENTO DE CIÊNCIA E TECNOLOGIA
INSTITUTO MILITAR DE ENGENHARIA
CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

1 Ten CARLOS ALBERTO DUARTE PINTO
1 Ten FELIPE AUGUSTO MARQUES DE ALCÂNTARA
1 Ten VINICIUS CARLOS OLIVEIRA DE ANDRADE

LABORATÓRIO AUTOMATIZADO PARA A ANÁLISE DE *MALWARES*

Rio de Janeiro
2017

INSTITUTO MILITAR DE ENGENHARIA

1 Ten CARLOS ALBERTO DUARTE PINTO
1 Ten FELIPE AUGUSTO MARQUES DE ALCÂNTARA
1 Ten VINICIUS CARLOS OLIVEIRA DE ANDRADE

LABORATÓRIO AUTOMATIZADO PARA A ANÁLISE DE *MALWARES*

Projeto de Fim de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientador: Maj Julio Cesar Duarte - D.Sc.

Rio de Janeiro
2017

INSTITUTO MILITAR DE ENGENHARIA

Praça General Tibúrcio, 80 – Praia Vermelha

Rio de Janeiro – RJ CEP: 22290-270

006.3 Pinto, Carlos Alberto Duarte

P659l Laboratório automatizado para a análise de malwares / Carlos Alberto Duarte Pinto, Felipe Augusto Marques de Alcântara, Vinícius Carlos Oliveira de Andrade; orientados por Julio Cesar Duarte – Rio de Janeiro: Instituto Militar de Engenharia, 2017.

66p. : il.

Projeto de Fim de Curso (PROFIC) – Instituto Militar de Engenharia, Rio de Janeiro, 2017.

1. Curso de Engenharia de Computação – Projeto de Fim de Curso. 2. Automatização. I. Alcântara, Felipe Augusto Marques de. II. Andrade, Vinicius Carlos Oliveira de. III. Duarte, Julio Cesar. IV. Título. V. Instituto Militar de Engenharia.

INSTITUTO MILITAR DE ENGENHARIA

1 Ten CARLOS ALBERTO DUARTE PINTO
1 Ten FELIPE AUGUSTO MARQUES DE ALCÂNTARA
1 Ten VINICIUS CARLOS OLIVEIRA DE ANDRADE

LABORATÓRIO AUTOMATIZADO PARA A ANÁLISE DE *MALWARES*

Projeto de Fim de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Instituto Militar de Engenharia, como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientador: Maj Julio Cesar Duarte - D.Sc.

Aprovado em 28 de setembro de 2017 pela seguinte Banca Examinadora:

Maj Julio Cesar Duarte - D.Sc. do IME - Presidente

Prof. Ricardo Choren Noya - D.Sc. do IME

Maj Humberto Henriques de Arruda - M.Sc. do IME

Rio de Janeiro
2017

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

FAMa	Framework de Aprendizado de Máquina
EPEx	Escritório de Projetos do Exército
URL	Unified Resource Location
API	Application Programming Interface
JSON	JavaScript Object Notation
NAT	Network Address Translation
HTML	HyperText Markup Language
MTV	Model Template View
MVC	Model View Controller
SQL	Structured Query Language
CSS	Cascading Style Sheets
MIT	Massachusetts Institute of Technology
NAT	Network Address Translation
XMLRPC	eXtensible Markup Language Remote Procedure Call
CIBR	Classless Inter-Domain Routing
PIL	Python Imaging Library

RESUMO

Análise de *malware* é um tópico de extrema importância nos dias de hoje. O fato da sociedade ser altamente dependente dos computadores e da internet, e dessa dependência crescer a cada dia, aliado ao aumento do número de *malwares* a cada ano torna todos os usuários bastante suscetíveis a eles. Nesse contexto é necessário que se desenvolvam métodos e ferramentas capazes de identificar essas ameaças.

Neste cenário, este projeto tem como objetivo criar um sistema automatizado para análise de *malwares*. O sistema recebe um artefato e o analisa utilizando diversas ferramentas tanto para análise estática quanto para análise dinâmica e ao final ele recolhe os relatórios de cada ferramenta e os apresenta ao usuário de modo a auxiliá-lo na classificação do artefato submetido.

O sistema desenvolvido neste trabalho será de grande valia na manutenção da segurança de ambientes acadêmicos e corporativos através da automatização de análise de artefatos. Através de uma aplicação distribuída na arquitetura cliente-servidor, há a possibilidade de disponibilizar acesso rápido e fácil a qualquer usuário que necessite analisar arquivos suspeitos.

ABSTRACT

Malware Analysis is a topic of utter importance these days. Due to the fact that society heavily relies on computers and on the internet, this dependency growing stronger every day, and the increase in the number of malwares, year after year, users become more vulnerable to them. In this context, it's necessary to develop methods and tools that are able to identify these threats.

In this scenario, this project's goal is to create an automated system for malware analysis. The system receives an artifact and analyzes it using tools for both static and dynamic analysis and, after processing it, the system forwards the reports generated by each tool to the end user, so as to help in the task of classifying the given artifact.

Due to the automation of artifact analysis, the final system developed after this work will be of great value in the security maintenance of academic and corporative environments. Through a distributed application, based on a client-server architecture, a quick and easy access may be provided to any user who needs to analyse suspicious files.

1 INTRODUÇÃO

Malware é todo tipo de software malicioso que executa atividades danosas na máquina de um usuário sem o seu conhecimento prévio. Dentre as principais atividades danosas, destacam-se o roubo de informações pessoais e confidenciais, a impossibilidade de que o usuário acesse a sua máquina ou a obtenção de vantagens financeiras. Os tipos de máquinas normalmente infectadas por *malwares* são computadores, *tablets* e *smartphones*.

A análise de *malwares* compreende um conjunto de ações a serem tomadas a fim de entender o comportamento de um artefato malicioso, as formas de identificá-lo e eliminá-lo, bem como ações defensivas que objetivam impossibilitar seus danos. A análise pode ser feita de duas formas: estática e dinâmica. Na análise estática, o estudo se baseia no código do *malware*, extraindo informações como dados em bases de antivírus, o *hashing* que funciona como uma espécie de "impressão digital" do *malware* e *Strings*, sem executar o código. Já na análise dinâmica, o código é executado e o estudo se baseia no comportamento do artefato malicioso. Apesar de existirem dois tipos de análises, elas não são, sozinhas, suficientes para compreender em sua totalidade o comportamento de um *malware*. Isso porque, diariamente, novos tipos de softwares maliciosos são desenvolvidos e introduzidos na rede e, para cada tipo, uma das análises pode se mostrar mais conclusiva ou as duas podem fornecer informações que se complementam.

Por ser uma atividade que oferece riscos de infecção, a análise dinâmica é feita em ambientes controlados, normalmente máquinas virtuais que simulam um ambiente para que o *malware* possa se manifestar e a partir daí extrair informações sobre o seu comportamento.

1.1 MOTIVAÇÃO

Segundo dados do instituto AV-TEST (AV-TEST, 2017), cerca de 390.000 novos *malwares* surgem diariamente e o número total deles cresce de ano a ano, como ilustrado na figura 1.1. Nela, é possível verificar que antes dos anos 2000, a quantidade de *malwares* era bastante reduzida e praticamente constante. Isso porque, nos anos anteriores, o computador ainda era um componente bastante caro e com poucas funcionalidades. Com o avanço do poder de processamento das máquinas, ocorreu um crescente barateamento do computador, tornando-o mais presente na vida do homem. Juntamente com o avanço

computacional, as redes de comunicação foram interligando cada vez mais o mundo como um todo. Com um mundo mais interconectado e uma quantidade crescente de usuários utilizando as facilidades providas pela Internet, o desenvolvimento de *malwares* se tornou uma atividade bastante atrativa para adquirir vantagens através de meios ilícitos.

Nesse contexto, os trabalhos que têm por finalidade coibir ações dessa natureza geram um grande impacto social ao fornecer segurança para o mundo digital.

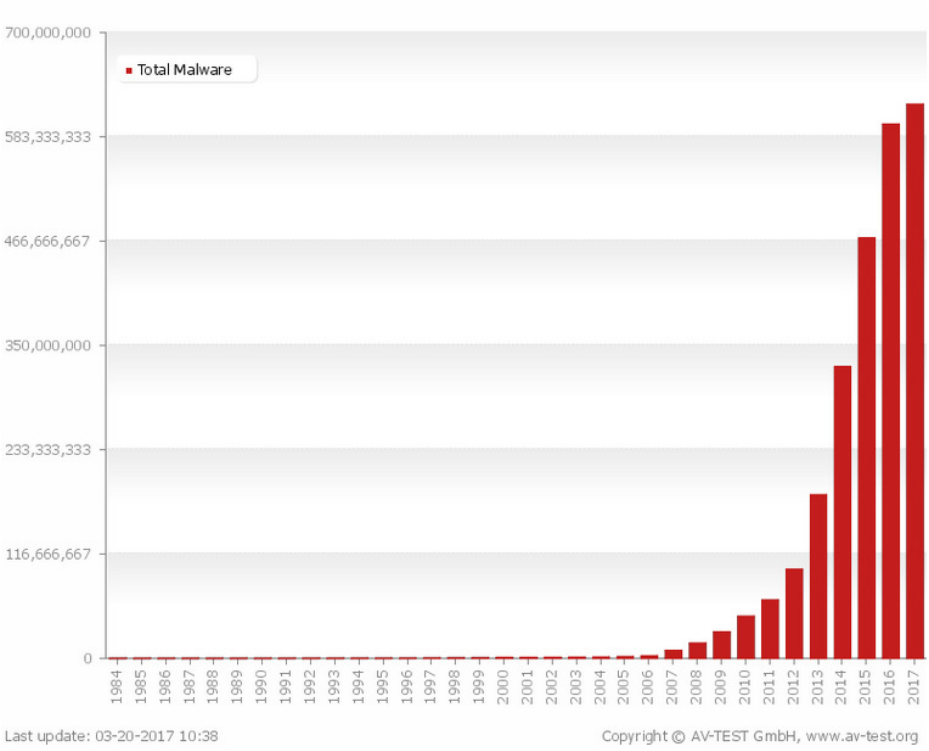

FIG. 1.1: Evolução do número total de *malwares*.

1.2 OBJETIVOS

O objetivo deste trabalho é implementar um sistema que seja capaz de realizar as diversas análises em um dado arquivo carregado pelo usuário, chamado aqui de artefato, auxiliando o processo de decisão de se determinar se o arquivo é um *malware* ou não. Estes arquivos serão carregados em uma aplicação *web* e o sistema conterá três componentes principais:

- Detecção por Antivírus: O artefato passa por uma varredura por diversos antivírus para detectar se já é um *malware* conhecido.
- Análise estática: O artefato é analisado com base somente em seu código executável.

- Análise dinâmica: O artefato é executado em ambiente controlado para que seu comportamento, ao ser executado, seja analisado.

Cada um destes componentes gerará um conjunto de relatórios que será disponibilizado para o usuário.

1.3 JUSTIFICATIVAS

Em 18 de dezembro de 2008, foi aprovado o decreto nº 6.703 que trata da Estratégia Nacional de Defesa. Esse decreto estabelece três setores estratégicos de defesa, que são: o nuclear, o cibernético e o espacial. Esses setores foram distribuídos entre as 3 forças, sendo o setor nuclear responsabilidade da marinha, o cibernético do exército e o espacial da aeronáutica.

A partir do momento em que o setor cibernético passou a ser capitaneado pela Força Terrestre, nasceu o Programa Estratégico de Defesa Cibernética que, em 2016, foi substituído pelo Programa Estratégico do Exército de Defesa Cibernética. Um dos objetivos desse programa é (EPEX, 2016):

> "Buscar inovações na área de Segurança da Informação e Comunicações, em especial a criptografia, por intermédio da estruturação de uma rede de laboratórios virtuais em instituições de pesquisas públicas e privadas nacionais, elevando a competência brasileira nesta área, ao patamar dos países mais desenvolvidos."

Nesse contexto, o desenvolvimento de um laboratório virtual que permita fazer análise de *malwares* alinha o presente trabalho com os interesses do Programa Estratégico do Exército de Defesa Cibernética, uma vez que contribui com inovações na área de Segurança da Informação.

1.4 ORGANIZAÇÃO DA DISSERTAÇÃO

A dissertação está estruturada da seguinte forma: no capítulo 2 é abordada a análise de *malwares*, identificando as formas mais comuns de análise que são encontradas na literatura. No capítulo 3 é feita uma descrição da proposta do trabalho com os diagramas utilizados para a modelagem do problema. No capítulo 4 é apresentada uma visão geral do *framework* escolhido para a implementação do laboratório virtual e das ferramentas

auxiliares empregadas. No capítulo 5 é descrito o processo de implementação da modelagem apresentada no capítulo 3 utilizando o *framework* apresentado no capítulo 4. No capítulo 6 é descrita a usabilidade do sistema do ponto de vista do administrador e do usuário final. No capitulo 7 é feita uma conclusão parcial do trabalho realizado.

2 ANÁLISE DE *MALWARE*

A análise de *malwares* tem como principais objetivos determinar o que um certo arquivo binário suspeito é capaz de fazer e como mensurar o dano que ele pode causar. Basicamente, existem duas técnicas de análise, a Análise Estática e a Análise Dinâmica.

2.1 ANÁLISE ESTÁTICA

Um *malware*, em geral, vem na forma de um executável. A análise estática consiste na inspeção desse código executável do artefato sem a necessidade de executá-lo e pode ser dividida em dois tipos (SIKORSKI; HONIG, 2012):

a) Análise Estática Básica: Quando há análise do executável sem leitura das instruções, baseando-se somente em seus atributos.

b) Análise Estática Avançada: Consiste em carregar o executável em um *disassembler* para analisar suas instruções.

Dois dos métodos mais utilizados para se realizar análise estática estão descritos abaixo (SIKORSKI; HONIG, 2012).

2.1.1 VARREDURA EM ANTIVÍRUS

Um dos primeiros passos da análise estática é a varredura em antivírus, pois há a possibilidade de que o artefato já tenha sido identificado. Entretanto, devido ao fato do antivírus depender de uma base de dados, de assinaturas de arquivos e de heurísticas, é fácil para novos *malwares* passarem despercebidos pelos mesmos.

2.1.2 ANÁLISE ESTRUTURAL

Na análise estrutural, o código e a estrutura do executável são analisados de forma a encontrar informações sobre suas funcionalidades. Da estrutura do arquivo podem ser extraídos alguns atributos do artefato, como momento de compilação, por exemplo. Outras informações são extraídas através de programas de depuração que analisam o código de máquina. Na tabela 2.1, são apresentados alguns dos principais atributos que podem ser extraídos de uma análise estática.

TAB. 2.1: Principais Atributos Extraídos numa Análise Estática

Atributo	Descrição
Hash	Identificação digital única do arquivo
Strings	Cadeia de caracteres com significado e escopo
Bibliotecas Importadas	Funções de sistema ou externas utilizadas pelo artefato
Empacotamento	Técnica utilizada para ofuscar o código malicioso
Entropia	Métrica para detectar empacotamento
Time-stamp	Data em que o código foi compilado
Anti-depuração	Técnica utilizada para detectar se um *debugger* está sendo utilizado
Funções Exportadas	Funções que o artefato exporta para uso de outros programas

2.2 ANÁLISE DINÂMICA

Na análise dinâmica, o objetivo é executar o *malware* e observar seu comportamento para extrair informações acerca de seus propósitos e danos a um sistema. Segundo (SIKORSKI; HONIG, 2012), a análise dinâmica pode ser subdividida em:

a) Análise Dinâmica Básica: consiste apenas em executar o *malware* e observar seu comportamento no sistema para remover a infecção, produzir assinaturas efetivas ou ambos.

b) Análise Dinâmica Avançada: envolve o uso de um depurador (*debugger*) para fazer uma análise mais detalhada do *malware*.

Por necessitar que o código seja executado, a análise dinâmica é uma atividade que oferece riscos e requer um ambiente controlado para que seja realizada. Para construção de um ambiente desse tipo, utilizam-se técnicas de virtualização que permitem simular diversos serviços que possivelmente estariam envolvidos nas atividades executadas pelo *malware*. A vantagem desse tipo de abordagem é que toda a análise acontece em um ambiente confinado, evitando a infecção da máquina do analista.

A identificação das atividades que o *malware* exerce ao infectar um sistema é feita a partir do monitoramento de um conjunto de características. Essas características são normalmente chamadas às APIs do Windows que tentam executar um processo a nível de *kernel* e que possivelmente tem por objetivo comprometer a infraestrutura do sistema. Não somente a presença da chamada da API, mas a quantidade de vezes que ela é chamada auxilia na identificação de amostras similares de *malwares* (FUJINO et al., 2015). Alguns exemplos de APIs encontradas na literatura (SIKORSKI; HONIG, 2012) e que foram utilizadas em outros trabalhos correlatos (MANGIALARDO, 2015) são:

- CreateFile: cria um arquivo ou abre um arquivo existente. Pode ser utilizado por um *malware* para despejar um arquivo no sistema.
- CreateMutex: cria uma exclusão mútua de objeto que pode ser usada por um *malware*.
- CreateProcess: cria e inicia um novo processo.
- CreateRemoteThread: utilizado para iniciar uma *thread* em um processo remoto. Um *malware* pode utilizá-lo para injetar código em um processo existente.
- CreateService: cria um serviço que pode ser iniciado em tempo de *boot*. Um *malware* pode utilizá-lo para persistência ou para carregar *drivers*.
- ReadFile: lê um arquivo.
- ReadProcessMemory: usado para ler a memória de um processo remoto. Regiões válidas de memória podem ser copiadas utilizando esta API. Um *malware* pode fazer uso desta API para obter, por exemplo, o número do cartão de crédito de um usuário, utilizando um código de busca.
- RegDeleteKey: apaga uma chave de registro e subchaves.
- RegEnumKeyEx: enumera as subchaves de um registro aberto. Pode ser utilizado em conjunto com RegDeleteKey para apagar recursivamente chaves de registro.
- RegEnumValue: enumera os valores para a chave de registro aberta.
- CreateSection: cria um objeto de seção.
- RegOpenKey: abre um *handle* para controlar a leitura e edição de um registro.
- ShellExecute: utilizado para executar um outro programa.

Uma lista mais completa das APIs pode ser encontrada no anexo 1.

3 LABORATÓRIO AUTOMATIZADO DE MALWARES

Este trabalho tem como proposta a criação de um sistema automatizado para análise de artefatos que gera relatórios que dão suporte a um analista na tarefa de determinação se um artefato é ou não um *malware*. A partir dos requisitos e das necessidades apresentadas, foi gerada uma modelagem lógica para o sistema de modo que o mesmo atinja seus objetivos.

3.1 REQUISITOS

O sistema possui uma série de requisitos obrigatórios que devem ser atendidos para que o sistema esteja completo, são eles:

- O sistema deve ser capaz de analisar *malwares* para a plataforma Windows de 32 bits;
- O usuário deve ser capaz de se cadastrar no sistema;
- O usuário, caso cadastrado, deve ser capaz de efetuar *login* no sistema;
- O usuário autenticado no sistema deve ser capaz de submeter um artefato para análise;
- O usuário autenticado no sistema deve ser capaz de acessar os relatórios das análises já finalizados;
- O sistema deve ser acessado pela *web*.

Existem ainda alguns requisitos opcionais, embora bastante desejáveis, que o sistema possua:

- Integração total com o FAMa, de modo a gerar um relatório final baseados nos relatórios das ferramentas e melhorar a capacidade de classificação do FAMa;
- O usuário poder modificar suas informações de cadastro, a senha e pedir um lembrete da senha;
- O usuário receber os relatórios conforme as ferramentas forem gerando-os e não apenas ao fim da análise;

- O usuário poder mudar as configurações das ferramentas de análise.

Baseado nesses requisitos foi criado um modelo lógico que visa atendê-los da forma mais plena possível e possua algumas características como robustez e escalabilidade.

3.2 MODELO LÓGICO

Baseado nos requisitos apresentados na seção anterior, assim como nos objetivos que o sistema deve alcançar e nas informações obtidas nas provas de conceito foi criado o modelo lógico apresentado na figura 3.1.

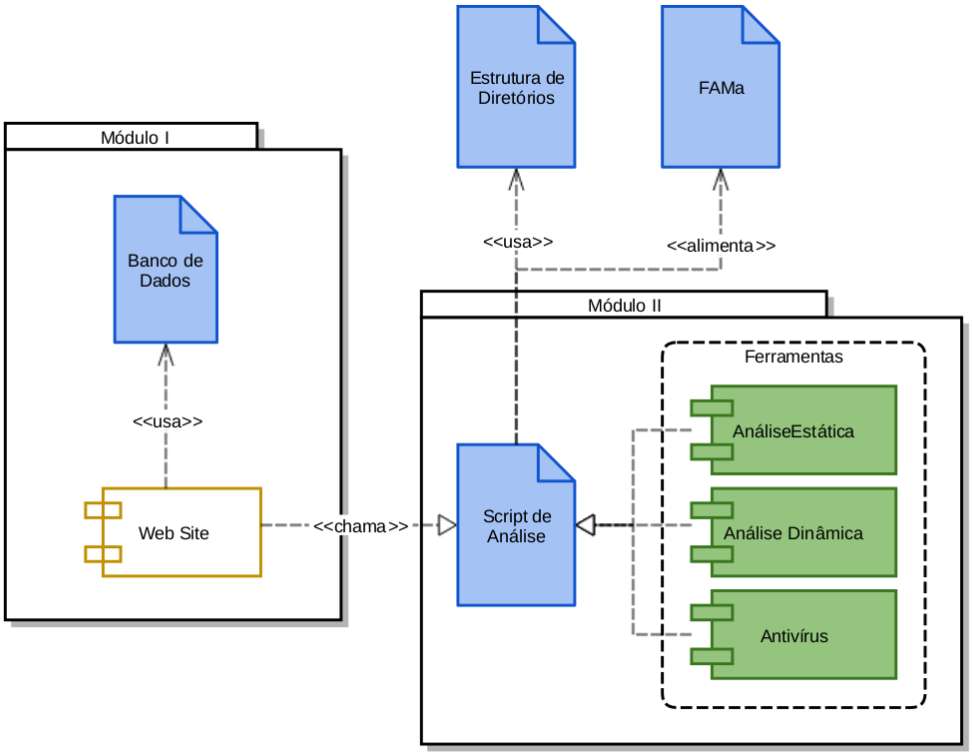

FIG. 3.1: Modelo lógico, descrição das partes do sistema e como interagem.

Nesse modelo o sistema é dividido em dois módulos, sendo o módulo I constituído do *Web Site* e do Banco de Dados e o II do Script de Análise. Os módulos funcionam de maneira independente e possuem como únicas relações o módulo I iniciar os processos do módulo II e a estrutura de diretórios comum.

O módulo I é responsável pelo gerenciamento das relações com o cliente, ou seja, é responsável pelo cadastramento do usuário no Banco de Dados e pela criação de seus diretórios dentro da estrutura, também é responsável pelo gerenciamento de *login* dos

usuários e pelo recebimento dos arquivos submetidos por ele. Ao receber o artefato este é transmitido ao módulo II para análise.

A análise e controle dos relatórios é feito pelo módulo II. Ao receber o artefato do módulo I, ele o envia para as ferramentas que fazem a análise. Cada ferramenta gera um relatório que é salvo no diretório do cliente para que possa ser apresentado para o mesmo posteriormente. Ainda é possível para o módulo II gerar um relatório centralizado e métodos para se relacionar com o FAMa.

Na figura 3.2, é apresentado o fluxo principal do sistema desde o recebimento do artefato enviado pelo usuário até o fim das análises e armazenamento dos relatórios. Esse é o modelo mínimo necessário para que o sistema funcione de forma satisfatória, porém é possível adicionar funcionalidades extras a esse fluxo, desde que não interfiram no funcionamento do mesmo.

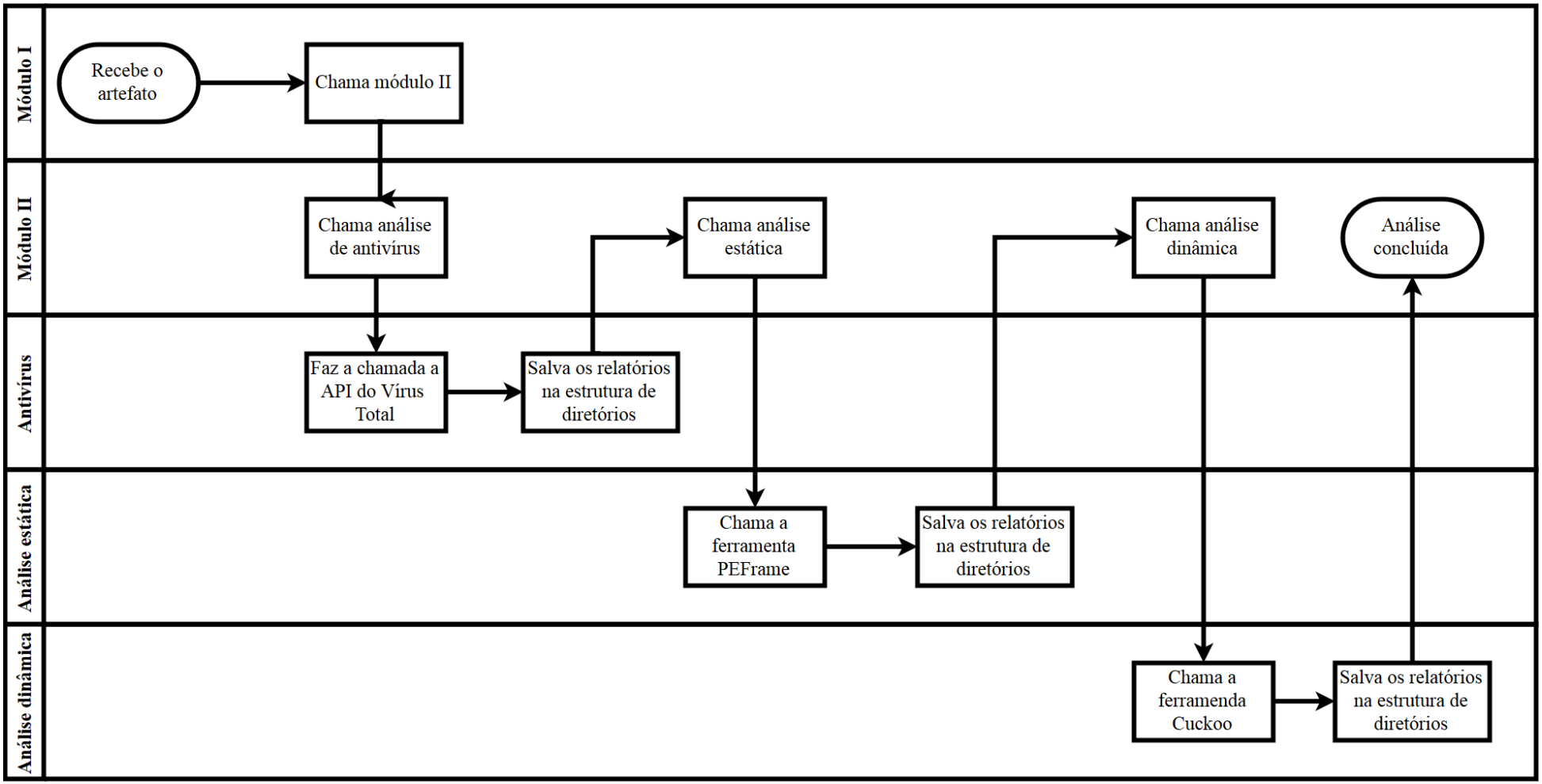

FIG. 3.2: Diagrama de atividades com raias, sequência do processo do recebimento ao fim da análise.

4 FERRAMENTAS UTILIZADAS

4.1 FERRAMENTA PARA DESENVOLVIMENTO WEB: DJANGO

A ferramenta escolhida para a confecção do *Web site* proposto no módulo I foi o *framework* Django. Os principais aspectos que motivaram essa escolha foram: a rica documentação do Django, propiciando uma vasta fonte de consulta; uma série de funcionalidades já prontas que o *framework* oferece, poupando o trabalho de "reinventar a roda" e o fato de ser implementado em Python, possibilitando uma melhor integração com as ferramentas dos analisadores estático e dinâmico.

Nessa seção, serão apresentadas algumas funcionalidades do Django que foram relevantes para a implementação do projeto até o momento.

4.1.1 O FRAMEWORK

Django é um *framework* de desenvolvimento *Web* implementado em Python. Tem como fundamentos o desenvolvimento rápido e pragmático de aplicações *Web*. Além disso, é gratuito e de código aberto (FOUNDATION).

Algumas características foram determinantes na sua escolha para o desenvolvimento do sistema proposto no capítulo anterior, dentre elas:

- Velocidade de desenvolvimento;
- Segurança: Além de ajudar a evitar falhas de segurança comuns, ele fornece um modo seguro de gerenciar contas de usuários e senhas;
- Funcionalidades prontas: Django vem com muitas funcionalidades já implementadas, como gerenciamento de arquivos estáticos, de usuários, interface de administrador, etc.

Django utiliza o padrão MTV *Model-Template-View*, uma variação do padrão MVC *Model-View-Controller*. As entidades do sistema (no caso deste trabalho, os usuários são as únicas entidades a serem armazenadas em banco de dados) são descritas como modelos, enquanto as páginas *Web* são vistas como *templates* e as relações entre as páginas, a lógica e os modelos são chamadas de *views*.

4.1.2 CARACTERÍSTICAS GERAIS

A seguir, é apresentada uma visão geral do Django, iniciando-se com conceitos básicos relativos ao funcionamento geral e finalizando com algumas funcionalidades que foram utilizadas na construção do presente trabalho.

4.1.2.1 CRIAÇÃO DE PROJETO E HIERARQUIA DE DIRETÓRIOS

Em Django, a criação de um projeto é automatizada através do comando:

```
$ django-admin startproject meu_projeto
```

O Django automaticamente cria a seguinte árvore de diretórios:

```
meu_projeto
├── manage.py
└── meu_projeto
    ├── settings.py
    └── urls.py
```

Cada arquivo serve a um propósito. Abaixo estão as descrições dos arquivos utilizados neste trabalho:

- `manage.py` contém uma série de funcionalidades para a administração da aplicação como, por exemplo, gerência de administradores, operações de banco de dados, operações para carregar o servidor, fornecimento de interface de linha de comando especializada, etc;
- `meu_projeto` é um diretório que serve como base para configurações e endereços;
- `settings.py` contém configurações do sistema *Web*, como lista de aplicações, configurações de banco de dados, de acesso e idiomas, etc;
- `urls.py` contém as declarações de endereços do sistema e o mapeamento destes endereços a funções, chamadas, neste contexto, de *views*.

Para iniciar o sistema no servidor, basta executar o seguinte comando:

```
$ python3 manage.py runserver
```

Este comando irá abir a porta 8000 do computador e o sistema poderá ser acessado através de `http://localhost:8000`. Porém, o sistema, no estado em que se encontra, não contém nenhuma aplicação. Uma aplicação é uma unidade de um projeto responsável por uma funcionalidade específica. Para criar uma aplicação, deve-se executar o seguinte comando:

```
$ python3 manage.py startapp meu_app
```

Um novo diretório para esta aplicação é gerado e alguns arquivos importantes também. O mais importante é o arquivo `views.py`. Este arquivo contém as funções, definidas em Python, correspondentes a cada URL definida em `urls.py`.

4.1.2.2 GERENCIAMENTO DAS APLICAÇÕES

Django oferece algumas ferramentas próprias para o gerenciamento da aplicação. Uma das mais importantes é a interface gráfica para administradores.

Administradores são usuários cadastrados através do comando

```
$ python3 manage.py createsuperuser
```

e têm capacidade de gerir o conteúdo das aplicações tendo permissão para incluir, alterar e deletar todas as instâncias de entidades *models* do sistema.

Outra ferramenta já embutida no *framework* é a gerência de banco de dados. O Django, quando instalado, utiliza o SQLite[1] para que não seja necessário instalar nenhum sistema de gerenciamento de banco de dados. Por meio da funcionalidade *Models*, a ser abordada na próxima seção, é possível criar tabelas e relações entre as entidades de uma maneira fácil e automatizada, sem a necessidade de se conhecer SQL.

Django tem um servidor próprio, leve e escrito em Python. Isso evita que o desenvolvedor tenha que se preocupar, enquanto estiver desenvolvendo, com detalhes de configuração de servidores. Este servidor é executado na porta 8000 do próprio computador.

Além disso, Django fornece uma interface de linha de comando contendo as APIs gratuitas fornecidas pelo próprio *framework*. Há inúmeras APIs, dentre elas APIs para gerenciamento do banco de dados, gerenciamento de arquivos, formulários, gerenciamento de modelos e templates, etc.

4.1.2.3 FUNCIONALIDADES

Dentre as funcionalidades disponibilizadas pelo Django e que foram largamente utilizadas neste trabalho, estão: as *models*, os *forms*, o diretório *template* e o diretório *static*. A seguir, é descrito como funciona cada uma delas.

- *Models*: um *Model* em Django trata basicamente da definição dos metadados e do armazenamento de suas instâncias. Cada definição de classe dentro do arquivo

[1] Disponível em: https://www.sqlite.org/

`models.py` será mapeada para uma tabela do banco de dados utilizado pelo desenvolvedor, poupando o tempo que o mesmo teria em formular a sequência de comandos para a criação do banco, para a manipulação das consultas e para a confecção de uma fábrica de conexões. No Django, cada atributo definido no *Model* representa um campo do banco de dados que será automaticamente gerado por uma API do *framework*;

- *Forms*: os formulários utilizados por cada aplicação dentro de um projeto em Django são definidos em um arquivo intitulado `forms.py`. Esse arquivo é composto de classes, onde cada classe define os campos relativos ao formulário utilizado pela aplicação. As instâncias dessas classes são feitas dentro da lógica de controle definida nas `views.py` e são renderizadas por um arquivo HTML que é passado como contexto;

- *Templates*: em Django, todos os arquivos HTML de uma dada aplicação são armazenados em um diretório denominado *templates* que fica dentro da própria aplicação. A finalidade desse diretório é indicar onde estão localizados os arquivos responsáveis por processar a informação a ser exibida na página. Outra característica também dos arquivos HTML desse diretório é que neles podem ser mesclados o código HTML com alguma lógica de controle que faça uso dos parâmetros passados dentro das funções definidas nas `views.py`. Os tipos de lógica utilizados nesse trabalho foram os laços de repetição e a manipulação de variáveis;

- *Static*: o diretório *static* tem a função de gerenciar todos o arquivos CSS, Javascript e JQuery utilizados pelos arquivos HTML que se encontram no diretório *templates*.

4.2 FERRAMENTAS PARA O FRONT-END

Para a confecção do front-end do *web site*, foram utilizadas as ferramentas mais consolidades em suas versões mais atuais. São elas:

- Javascript: é uma linguagem de programação que controla o HTML e o CSS de uma página para manipular seu comportamento;

- JQuery: é uma bibioteca de Javascript que auxilia na manipulação de documentos HTML, gerenciamento de eventos, dentre outras funcionalidades;

- Bootstrap[2] v3.3.7: é um *framework* de HTML, CSS e JS para desenvolvimento responsivo e projetos do tipo *mobile first*;

[2]Disponível em: http://getbootstrap.com/about/

- CSS: é uma linguagem para definição de estilo dos componentes de um documento HTML;
- Fine-Uploader 5: é uma biblioteca para fazer *upload* de arquivos escrita em JavaScript e que possui uma série de funcionalidades, tais como: barra de progresso, recurso *drag-and-drop*, validação do tipo de arquivo, dentre outras;

Para fazer uso dessas ferramentas foram empregados 2 *templates* gratuitos de *sites*. Um deles está sendo utilizado para a interface de *login* e de registro dos usuários e é disponibilizado por AZMIND[3]. O outro é um *dashboard* utilizado para gerenciar a exibição do conteúdo fornecido para o usuário, sendo disponibilizado pelo MIT[4] também de forma gratuita.

A vantagem de utilizar esses *templates* gratuitos é que a customização de seus componentes para um uso específico demanda menos tempo para o desenvolvimento que construir todo o *front-end* do início, permitindo que seja dada mais ênfase no *back-end* da aplicação que realmente implementa os requisitos necessários e desejáveis para o seu bom funcionamento.

4.3 FERRAMENTA PARA ANÁLISE ESTÁTICA: PEFRAME

Existem inúmeras ferramentas para análise estática disponíveis. Dentre todas as mais famosas, três foram selecionadas para estudo: Exeinfo PE[5], PEview[6] e PEframe[7] A escolha da ferramenta para análise estática foi feita baseada em dois requisitos internos da aplicação:

a) Facilidade de carregar o artefato na ferramenta;

b) Facilidade de recuperação do relatório.

Ambos a Exeinfo PE e a PEview funcionam em ambiente Windows, suas interfaces são gráficas e interativas e não são de código aberto. Estas características dificultam o seu uso de maneira automatizada pelo sistema em desenvolvimento. Além disso, a aplicação Web está sendo executada em um servidor cujo sistema operacional é baseado em Linux.

[3]Disponível em: http://azmind.com/bootstrap-login-register-forms-templates/
[4]Disponível em: http://usebootstrap.com/themes/admin-dashboard
[5]Disponível em: http://exeinfo.atwebpages.com/
[6]Disponível em: http://wjradburn.com/software/
[7]Disponível em: https://github.com/guelfoweb/peframe

Com base nisto, a ferramenta selecionada foi o PEframe. PEframe é uma ferramenta de código aberto para a análise estática de Portable Executables e outros arquivos genéricos. Todos os atributos listados na tabela 2.1, na página 15, com exceção da entropia e de *strings*, compõem o relatório gerado pelo PEframe. Apesar disso, a ferramenta dá informações sobre empacotamento e disponibiliza uma opção para a extração de *strings*.

PEframe pode ser executado por linha de comando em ambiente Linux e retornar o relatório em formato JSON. Estas foram as razões para a sua escolha como ferramenta mais apropriada.

O comando em interface de linha de comando para executar o PEframe e salvar o relatório em um arquivo JSON é o seguinte:

```
$ peframe --json <executavel>
```

Já o comando para extrair as *strings* é o seguinte:

```
$ peframe --strings <executavel>
```

4.4 FERRAMENTA ONLINE DE VARREDURA POR ANTIVÍRUS: VIRUSTOTAL

O VirusTotal é uma ferramenta online que analisa arquivos e URLs permitindo a identificação de conteúdos maliciosos por *softwares* antivírus e escaneadores de URLs, possibilitando também a identificação de falsos positivos (VIRUSTOTAL, 2017).

O VirusTotal é um serviço livre e possui uma API pública para escanear arquivos e acessar relatórios. A API pode ser utilizada pela construção de *scripts* simples, fornecendo relatórios no formato JSON e sem que seja necessária a utilização da interface *Web* do VirusTotal. Essas foram as principais razões que motivaram a escolha da ferramenta para o uso de varredura por antivírus.

Para fazer uso da API, é necessário apenas ter uma conta no VirusTotal. Ao criar uma conta, uma chave é gerada. Essa chave é utilizada como um atributo a ser configurado nas requisições de arquivos submetidos. Um dos requisitos para que seja feito o uso em conformidade com os termos do serviço é que a API seja utilizada apenas para fins não comerciais e que não funcione como uma substituta para os produtos antivírus. Para coibir usos indevidos, o VirusTotal limita o número de requisições a quatro por minuto. Para resolver esse problema foi implementada uma fila simples onde o sistema consegue acesso a um arquivo, que não pode ser acessado por mais de 1 processo por vez, espera quinze segundos e depois faz a requisição, liberando o arquivo ao final.

Para a obtenção do relatório do VirusTotal, é necessário executar os seguintes coman-

TAB. 4.1: Informações da API Virus Total para cada antivirus

Atributo	Descrição
detected	Informa se o antivirus detectou o artefato submetido como um *malware*
version	Versão do antivirus consultado
result	Resultado da análise feita pelo antivirus
updated	Data de atualização do antivirus

dos:

```
$ python virusTotal.py <nome_do_arquivo>
```

virusTotal.py é um *script* em Python reponsável pela requisição para o servidor do VirusTotal responsável pela análise. Seu código pode ser visto no apêndice 6.

A lista de informações fornecidas pela API é exibida na tabela 4.1.

4.5 FERRAMENTA PARA ANÁLISE DINÂMICA: CUCKOO SANDBOX

O Cuckoo Sandbox é um sistema de análise dinâmica de *malwares*. Nele, um ambiente isolado é criado para permitir que o código de um determinado artefato seja executado e, a partir da execução, um relatório é gerado com informações referentes ao comportamento e atividades do artefato.

Dentre as principais razões que motivaram a escolha dessa ferramenta para a análise dinâmica, estão:

- *Software* livre;
- Executa em ambiente Linux;
- Possibilidade de fazer customizações;
- Interface em linha de comando.

Para o correto funcionamento do Cuckoo, uma série de etapas devem ser configuradas antes que uma submissão de um arquivo seja feita. Uma das etapas é a instalação de um *software* virtualizador para criar um ambiente confinado. O *software* escolhido foi o VirtualBox porque dentre as demais opções fornecidas pelo Cuckoo, é o que possui o maior número de funcionalidades implementadas, além de possuir a interface VBoxManage que dá suporte a linha de comando.

Após a instalação do software virtualizador, uma máquina virtual deve ser criada para que o *malware* seja posto em execução. O sistema operacional da máquina é escolha do analista, no caso do presente trabalho foi escolhido o sistema Windows XP. Após a

criação da máquina virtual, todas as opções de *firewall* e antivirus devem ser desabilitadas para que a máquina esteja em uma condição mais vulnerável possível para extrair todas informações do comportamente do *malware*. Além disso, dentre as opções de adaptadores de rede fornecidas pelo VirtualBox (NAT, *bridge*, *host-only* e *internal network*), a opção *host-only* deve ser selecionada para impossibilitar que exista tráfego de rede fora do ambiente confinado e para possibilitar a troca de arquivos entre a máquina que está fazendo a análise e a máquina a ser analisada. Para que essa troca seja possível, uma pasta deve ser criada na máquina que fará a análise e adicionada nas configurações de pasta compartilhada da máquina virtual com o ambiente confinado.

Após os procedimentos elencados acima, o *script* `agent.py` disponibilizado pelo Cuckoo deve ser colocado na pasta compartilhada e executado na máquina a ser analisada. Esse *script* atua basicamente na troca de dados e na comunicação, alimentando um servidor XMLRPC que estará aguardando por conexões. Depois disso, deve-se tirar um *snapshop* da máquina que será utilizada para executar o *malware*. Esse *snapshop* será utilizado toda vez que uma análise for submetida ao Cuckoo.

Para configurar os resultados que serão exibidos pelo Cuckoo, aplicações específicas devem ser adicionadas ao *snapshot*. Exemplos de aplicações que podem ser adicionadas são: Wireshark e Volatility. A primeira permite que o sandbox inclua no relatório o tráfego de rede gerado e, a segunda, a análise do *dump* de memória. Além dos resultados que podem ser customizados pela inclusão de aplicações na máquina virtual, o Cuckoo inclui outros resultados que fazem parte da análise padronizada. Dentre eles estão: informações de controle como o nome do arquivo, *hash* e o tamanho; sumário do comportamento incluindo a enumeração dos diretórios, registros e outros campos; relatório do VirusTotal. Todas essas informações são disponibilizadas em dois formatos: HTML e JSON. Um exemplo da saída com informações de controle é exibida na figura 4.1. A amostra utilizada nessa análise foi do vírus `Autorun.inf`. Um extrato da saída no formato JSON é encontrado no apêndice 4.

File Details

File name	autorun.inf
File size	3553 bytes
File type	ASCII text, with CRLF line terminators
CRC32	CF68E7D9
MD5	63289b78dedd08b38bf32a5542c129e3
SHA1	42ed7dfb280bf27c6578c972594e2b52fddfcace
SHA256	d442397c29ab7fdfc897af3f3c2bc4f976eba0d852df4c4dfe621ecf34410198
SHA512	786ae1523804f860e463639f4de6709dc287e3433e71b847b76aade031cc70aa2c564217da850e54abcc2593ad033e990587427811208deb7db51b490c826866
Ssdeep	None
PEiD	None matched
Yara	None matched
VirusTotal	Permalink VirusTotal Scan Date: 2017-04-25 23:08:59 Detection Rate: 13/54 (Expand)

FIG. 4.1: Saída da interface *web* do Cuckoo.

5 DESENVOLVENDO O LABORATÓRIO DE MALWARES

A partir do modelo lógico, foram implementados os módulos I e II descritos na seção anterior. O módulo II foi completamente separado do I tornando-se independente, sendo apenas chamado por ele.

5.1 MÓDULO I

Para a implementação do primeiro módulo, que consiste na interface *web* com o usuário, foram criados dois *apps* sendo um responsável pelo gerenciamento dos dados relacionados com os usuários (*manageuser*) e outro responsável pelo gerenciamento do que diz respeito aos arquivos (*managefiles*). Além dos dois *apps* também são usados para o módulo I algumas funções definidas em *src.definitions.py*, apresentado no apêndice 5.

5.1.1 MANAGEUSER

O primeiro *app* é o responsável pelo gerenciamento de tudo que é relacionado com a criação e a manutenção dos usuários. Para que consiga cumprir da melhor forma o seu papel, ele se utiliza dos sistemas de *login* e registro de usuários já existentes no Django de forma a garantir maior segurança e eficiência.

Para que o sistema de registro em banco de dados do Django fosse utilizado, foi necessária a adição do campo que guarda as informações de diretório do usuário, sendo então necessária a criação da classe `Directories` dentro do arquivo `models.py` (Apêndice 2). Essa classe é utilizada pelo Django para que ele modifique seu banco de dados, adicionando os atributos da classe como campos em uma nova tabela do banco, e, por meio de um método definido nela, preencha o campo automaticamente no momento que um usuário é criado, não necessitando que seja feito separadamente e de forma manual. Como último passo, foi preciso criar uma relação um para um com os campos da tabela de usuários padrão do Django, o que foi feito dentro do arquivo `admin.py`.

Uma vez que a base para a criação de usuários foi estabelecida, torna-se necessária apenas a criação dos formulários, de forma que os usuários possam fazer as requisições. Foram criados quatro formulários, classes definidas em `forms.py`, como visto no Apêndice 3, e que herdam de `django.forms.Form`, sendo um para registro, o segundo para *login*, um para alteração de informações do usuário e um para a senha. Definindo os formulários

dessa forma, permite-se que sejam usadas as funções nativas do Django para validação dos mesmos além de facilitar a customização dos estilos sem se preocupar com eles.

Finalmente, basta que as requisições dos usuários sejam tratadas pelas funções definidas dentro de `views.py`, mostradas no Apêndice 1. A requisição mais simples é a de *logout*, que simplesmente chama a função nativa do Django para permitir a saída do usuário do sistemas. As demais, *register*, *login*, *update_infos* e *change_password*, possuem uma estrutura bastante parecida pois são feitas para que se evitem erros, sem atrapalhar seu funcionamento particular.

Inicialmente é verificado se o método da requisição é do tipo POST e, em caso positivo, um objeto da classe do formulário correspondente é instanciado com os dados recebidos. Após isso o objeto é utilizado para verificar se o formulário foi preenchido de uma forma válida e é feita a lógica específica de cada função, que pode ou não redirecionar para outra página. Caso o método não tenha sido POST um formulário vazio é instanciado. Se a função não redirecionar o usuário para uma nova página ele é mantido na página em que está, porém com as informações já preenchidas no formulário.

De forma mais específica para cada uma delas a função *register* verifica se o usuário já existe e se a senha e sua confirmação são iguais. Caso tudo esteja correto, ele registra o novo usuário, o autentica e redireciona para a página principal. Além disso a função de registro também é responsável pela criação do diretório principal do usuário dentro da estrutura. A função *login* apenas autentica o usuário e o redireciona para a página principal. Na *update_infos* é apenas feita a atualização das informação e na *change_password* é verificado se a senha está correta e se a nova senha corresponde com a confirmação e, em caso positivo, atualiza a senha do usuário e o mantém autenticado.

5.1.2 MANAGEFILES

Esse *app* é responsável pelos processos relacionados com os arquivos e derivados dele, ou seja, ele é basicamente responsável por receber o arquivo do usuário, fazer a chamada do módulo II para que o arquivo seja analisado e posteriormente seja feita a apresentação dos relatórios decorrentes das análises.

Quando um arquivo é enviado para o *app* uma requisição para o servidor é criada dentro de um script no arquivo `forms_file_uploader.html` que se encontra dentro da pasta *templates* do *app managefiles*. Essa requisição é feita através da URL `/up` e tem por propósito manipular parâmetros do Fine-Uploader 5. Ao receber essa requisição, a classe `UploadView` definida em `managefiles/views.py` é responsável por chamar a

função *save_upload_file*, definida em `src/definitions.py` que trata do armazenamento dos arquivos enviados pelo usuário, gerando dentro do diretório do usuário, o diretório referente à análise em que é salvo o arquivo enviado e um outro diretório onde serão armazenados os relatórios, concluindo com a chamada ao script do módulo II. Após isso, um objeto JSON é enviado pelo servidor indicando se o arquivo enviado foi salvo ou não, sendo esse objeto enviado pela chamada da função *make_response* definida também em `managefiles/views.py`. Dependendo da resposta do servidor, uma ação é tomada pelo *script* que gerencia o comportamento da página, indicando ao usuário o *status* de sua submissão.

Finalmente, o *app* possui as funções *show_directories* e *download_files* que juntas servem para que o usuário possa ter acesso aos relatórios. A primeira função lista todas as análises referentes aos arquivos enviados pelo usuário, juntamente com o nome dos arquivos submetidos e os *links* para os relatórios gerados. A segunda, possibilita fazer o *download* do relatório selecionado ou exibi-lo no *browser*.

5.2 MÓDULO II

O módulo II é o responsável pelo gerenciamento das ferramentas e das análises, bem como formatar e salvar os relatórios no local correto. Funciona com um script principal que faz a chamada das ferramentas de modo a manter independentes tanto a análise quanto a apresentação dos relatórios ao mesmo tempo que mantém o gerenciamento centralizados.

O módulo II conta com três funções que são chamadas pela parte principal do script `src.tools.py`, onde também são implementadas. As funções *generate_static_reports*, *generate_virus_total_reports* e *generate_dynamic_reports* chamam, respectivamente, a ferramenta de análise estática, o VirusTotal e a ferramenta de análise dinâmica, sendo responsáveis por gerenciar os relatórios.

A função *generate_static_reports* faz uso de funções do módulo *subprocess* que permite que sejam utilizados comandos de *shell script* através do Python. Ao receber o retorno do comando, que é o relatório da análise, o salva em uma variável para que possa ser formatado. Após a formatação do relatório ele é salvo em um arquivo para que posteriormente possa ser apresentado ao usuário.

No caso da função *generate_virus_total_reports* a implementação é bem mais simples uma vez que o VirusTotal possui uma API própria para Python. Sua implementação consiste em fazer uma chamada a API e formatar a resposta. Uma vez que a resposta seja formatada, é gerado um arquivo contendo o relatório.

Na função *generate_ dynamic_ reports* são feitas duas requisições para a API do Cuckoo sandbox. Uma delas é responsável por submeter para a análise um arquivo enviado pelo usuário, retornando um objeto JSON com um identificador a ser utilizado para armazenar os arquivos dentro de uma pasta do sandbox. A outra requisição é responsável por consultar o servidor para determinar o *status* da análise, sendo as repostas possíveis: `'pending'`, `'running'`, `'completed'` e `'reported'`. Essa requisição é feita em intervalos de 60s e, quando a resposta obtida é `'reported'`, a pasta com todos os arquivos da análise dinâmica é copiada para o diretório do usuário e disponibilizada para *download*. Além disso, é copiado também o arquivo HTML referente a análise para ser exibido no *browser* em um formato mais amigável para o usuário.

5.3 INTERFACE COM O USUÁRIO

Para cada um dos *app* do módulo I, descritos anteriormente, foi confeccionada uma interface para a comunicação com o usuário. Essa interface busca manter a familiaridade com recursos *web* amplamente utilizados na atualidade, tais como a opção de fazer *upload* de arquivo através de um botão ou simplesmente por pegar e arrastar para uma área delimitada e a opção de utilizar contas vinculadas em outros serviços para fazer *login*. Mais detalhes das interfaces são apresentadas no próximo capítulo.

Além disso, foi dado o nome de artefAthos ao sistema, como uma forma de criar no usuário um mnemônico que associe o ato inicial de interação com o sistema (submeter um artefato) e o gigante Athos que, na mitologia grega, chegou a derrotar Zeus. No contexto dos *malwares*, Zeus também é o nome de um cavalo de tróia que recebe outras denominações como ZeuS ou Zbot, capaz de executar atos criminosos e roubar informações bancárias. Sendo o propósito do presente trabalho combater e coibir crimes dessa natureza, o gigante Athos foi utilizado como uma metáfora para dar nome ao sistema.

6 PORTABILIDADE E USO DO SISTEMA

A usabilidade do sistema pode ser vista através das perspectivas do administrador, responsável pela implantação e manutenção, e do usuário, que simplesmente faz uso do mesmo. Serão abordadas nas próximas seções os passos necessário tanto para implantação como também para uso do sistema.

6.1 IMPLANTAÇÃO E INICIALIZAÇÃO DO SISTEMA

Para a implantação do sistema é preciso inicialmente ter todas as suas dependências instaladas. São elas:

- Python 3
- Django
- django-widget-tweaks
- requests
- json2html
- PEFrame
- VirtualBox
- Cuckoo

Além dos listados acima, o Cuckoo e o PEFrame tem suas próprias dependências, já mencionadas no capítulo referente as ferramentas utilizadas.

Para a configuração da parte relativa a análise dinâmica, é necessário garantir que os arquivos de configuração do Cuckoo estejam devidamente definidos com as informações das máquinas virtuais a serem utilizadas, a saber: o nome das máquinas, os sistemas operacionais, os IP's e os respectivos *Snapshots* a serem utilizados para inicializar as máquinas. Abaixo, estão descritos os arquivos de configuração e o propósito de cada um:

- `cuckoo.conf`: define as opções de análise e parâmetros relativos ao comportamento do sandbox;

- `auxiliary.conf`: permite habilitar módulos adicionais;
- `<machinery>.conf`: define opções relativas ao *software* de virtualização utilizado;
- `memory.conf`: utilizado para a configuração do Volatility;
- `reporting.conf`: define os tipos de relatórios gerados e suas extensões.

Como o Cuckoo não foi projetado para o uso de um protocolo de atribuição dinâmica de IPs, tal como o DHCP, é necessário atribuir endereços estáticos para cada uma das máquinas virtuais que serão utilizadas nas análises. Esse requisito do Cuckoo impossibilita que seja automatizado o processo de geração de novas máquinas de acordo com a capacidade dos usuários e requisições, uma vez que cada máquina clonada teria o mesmo IP da máquina original, resultando em um problema de comunicação na rede. Uma solução para esse problema da falta de automação na quantidade de máquinas em operação é o uso de distribuições probabilísticas que modelem os processos de chegada dos usuários e o atendimento de suas requisições, permitindo dimensionar a quantidade de máquinas que seja ideal de acordo com um determinado horário ou pico de demanda. Por ser algo que depende da operação do sistema, essa solução não será aprofundada nesse trabalho.

Para a configuração de cada máquina virtual, é descrito brevemente um roteiro a ser seguido. Como nesse trabalho foi utilizado o VirtualBox como *software* virtualizador e o Windows XP como sistema operacional para a máquina a ser utilizada para fazer a análise, o roteiro a seguir é feito com os *softwares* empregados durante o desenvolvimento.

Procedimentos:

- O primeiro passo para garantir que o Cuckoo se comunique com a máquina virtual é que o adaptador de rede *host-only* seja selecionado e que a interface desse adaptador seja configurada, atribuindo-se um IP, conforme ilustra a figura 6.1;

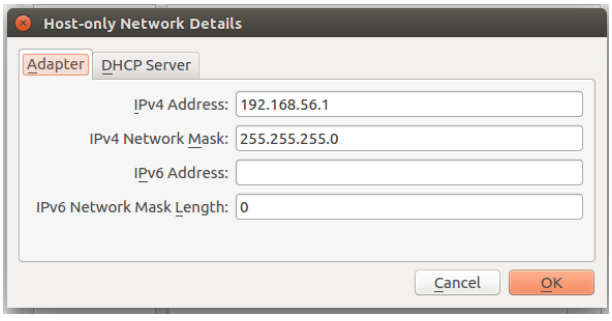

FIG. 6.1: Atribuição de IP à interface *Host-Only*

- Após isso, é necessário configurar uma rede local para a máquina virtual, atribuindo para a mesma um IP estático compatível com a faixa CIDR configurada na interface do adaptador *host-only*, como ilustra a figura 6.2. Uma boa prática para verificar a conectividade é executar o comando `ping` no *shell* do hospedeiro e da máquina virtual. Caso as requisições do comando sejam recebidas, a configuração está correta e existe um canal de comunicação entre a máquina hospedeira e a máquina virtual;

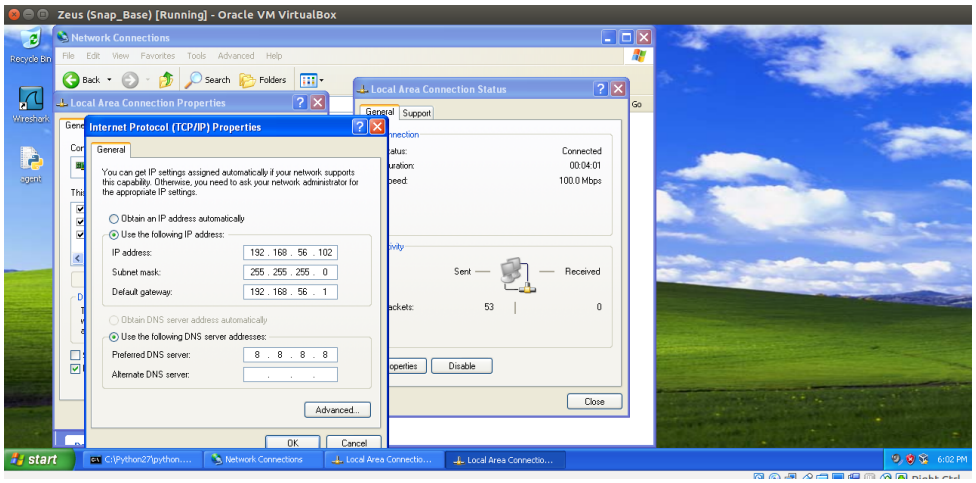

FIG. 6.2: Atribuição de IP estático no Windows XP

- Com a configuração de rede pronta, o próximo passo é instalar na máquina programas que sejam responsáveis pela execução dos arquivos submetidos ou que tragam informações a respeito do comportamento do sistema. No caso do presente trabalho, foram instalados o Wireshark e o PIL. O primeiro permite obter informações acerca do tráfego de rede gerado durante a análise e, o segundo, permite que sejam feitos *screenshots* durante a análise. Qualquer utilitário que seja instalado na máquina e seja devidamente habilitado nos arquivos de configuração do cuckoo permitirá que o sandbox extraia informações relativas ao respectivo utilitário, gerando um arquivo correspondente na pasta de relatórios;

- Com a configuração da rede e dos utilitários instalados, resta apenas incluir o *script* `agent.py` na máquina virtual, executá-lo e retirar o *snapshot* da máquina para guardar o estado a ser utilizado para as futuras análises. A figura 6.3 ilustra a execução do agente. Apesar de não ser impressa nenhuma mensagem no *shell* do Windows, o agente está funcionando em *background*. Com isso, encerra-se a configuração da máquina virtual.

Com todas as dependências instaladas e o sandbox devidamente configurado, é pos-

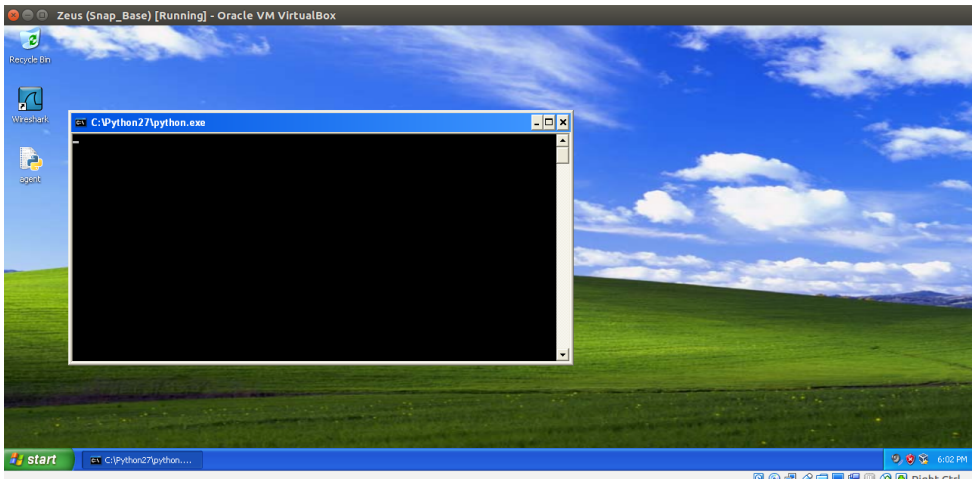

FIG. 6.3: Execução do script agent.py

sível prosseguir com a implantação do sistema. Para isso basta copiar a pasta principal contendo todos os arquivos do sistema para o local desejado e configuar o servidor *web* do Django com as informações desejadas. Com isso, o sistema já está pronto para uso, precisando apenas ser iniciado.

Para se inicializar corretamente o sistema é preciso que 4 componentes sejam inicializados: as máquinas virtuais, o Cuckoo, o servidor da API do Cuckoo e o servido *web* do Django. Para facilitar essa inicialização foi criado o script *run* que faz a inicialização de cada um deles e já redireciona sua saída para seu respectivo arquivo de log.

Após os passos acima o sistema estará devidamente funcional e poderá ser acessado pelos usuários. A partir dessa configuração inicial, o sistema funciona sem a necessidade de intervenções externas por parte do administrador.

6.2 UTILIZAÇÃO DO SISTEMA

Do ponto de vista do usuário, a utilização do sistema é intuitiva pois segue padrões adotados por diversas páginas na internet. Quando o usuário não estiver autenticado e for acessar alguma página ele será automaticamente redirecionado para a página apresentada na figura 6.4. Nela foram colocados os formulários de *login* e registro, para usuário que já possuem e que não possuem cadastro, respectivamente.

Já a figura 6.5 exibe a interface que se segue após o usuário efetuar *login* ou criar uma conta através do formulário de registro. Nela, foi colocada uma barra na lateral esquerda com as opções *home* e relatórios. Ainda que sejam poucos os campos do *menu dashboard*, com o crescimento do sistema, será possível acoplar mais opções nesse *menu*, tornando-o mais rico. Além disso, a barra de busca que consta na parte superior serve para que a

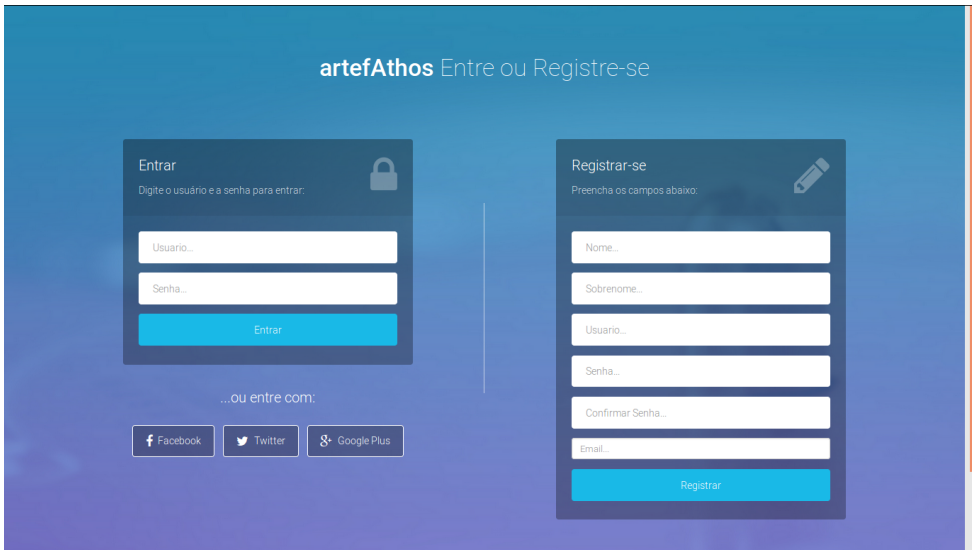

FIG. 6.4: Tela inicial

busca por um relatório específico seja feita sem ter que recorrer a ordem cronológica na página de relatórios, permitindo aprimorar a experiência de usabilidade do sistema.

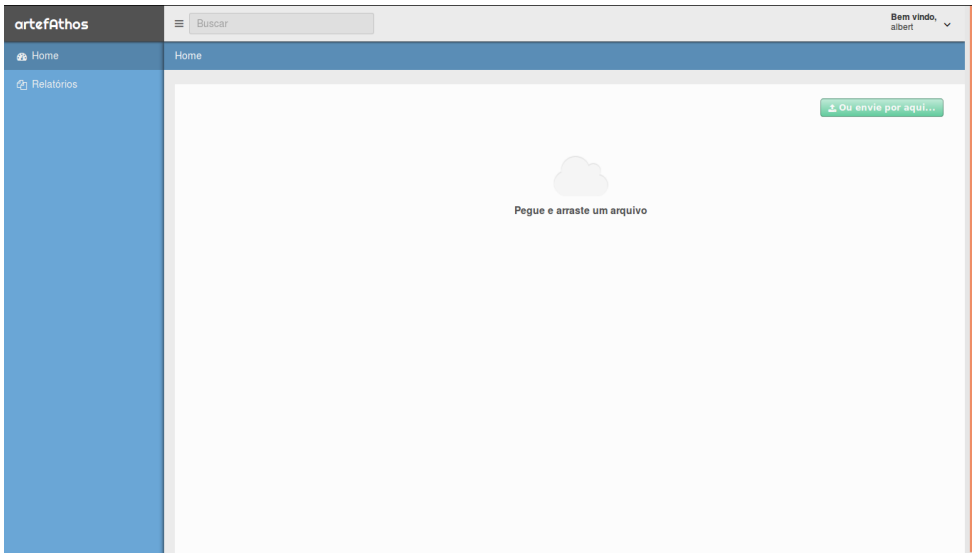

FIG. 6.5: *Dashboard* do usuário

Ainda na figura 6.5, é possível visualizar a região central que permite que os artefatos sejam submetidos através do recurso *drag-and-drop* ou pelo uso do botão verde no canto superior direito. Ao submeter um arquivo por quaisquer um desses recursos, as função geradoras dos relatórios (*generate_virus_total*, *generate_static_reports* e *generate_dynamic_reports*) são chamadas. Há também, no canto superior direito da interface mostrada na figura 6.5, um pequeno recado de boas vindas para o respectivo usuário da sessão juntamente com um botão que dá suporte aos recursos de *logout*, alterar senha e

alterar dados cadastrados.

Por meio do *menu* na barra da esquerda é possível acessar a página inicial, figura 6.5, através da opção *home* ou a página dos relatórios, figura 6.6 apresentada a seguir, através do botão opção relatórios. Na página de relatórios é apresentada uma lista com todas as requisições feitas pelo usuário e os botões para que sejam apresentados os relatórios referentes a cada ferramenta. Enquanto o relatório não está pronto para ser exibido, um botão vermelho é exibido. A medida que os relatórios vão sendo gerados e estão prontos, o botão vermelho é substituído por outros dois, um que permite ao usuário baixar a versão bruta do relatório e outro para acessar a versão tratada, feita para facilitar a leitura.

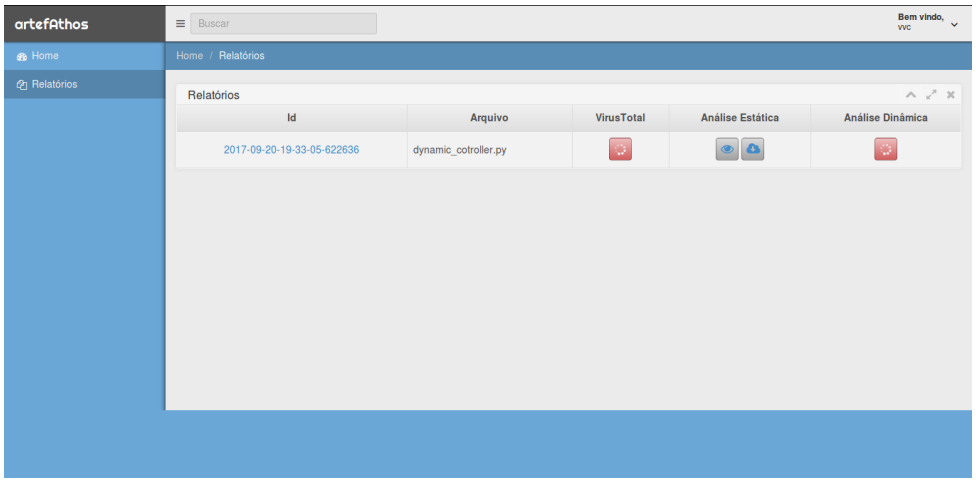

FIG. 6.6: Página de relatórios

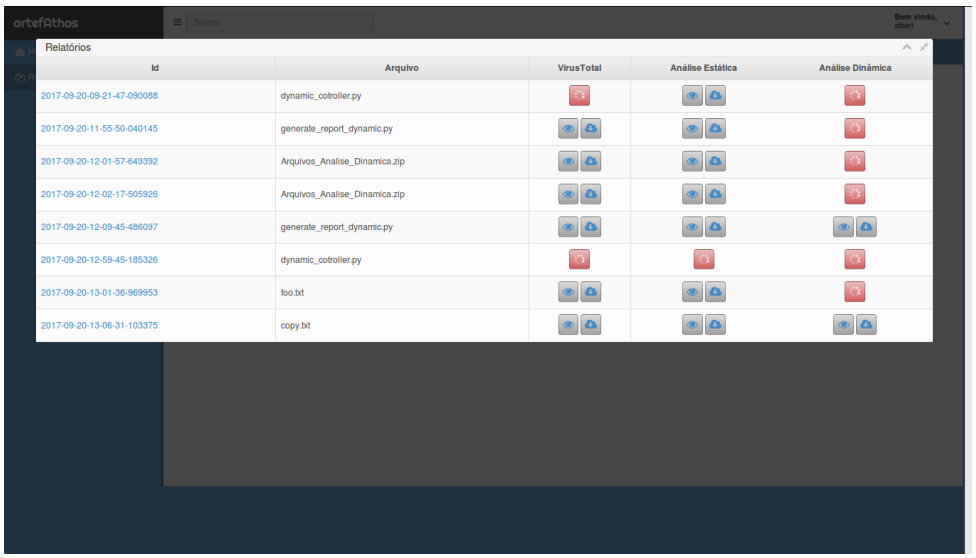

FIG. 6.7: Página de relatórios expandida

No canto direito do cabeçalho existe um botão que exibe um menu com as opções perfil, configurações e sair. A opção sair apenas remove o acesso do usuário ao sistema.

Sobre as demais opções, a opção “perfil” leva para a página da figura 6.8 onde o usuário pode atualizar suas informações de cadastro e a opção "configurações"direciona para a página da figura 6.9 que permite ao usuário alterar sua senha.

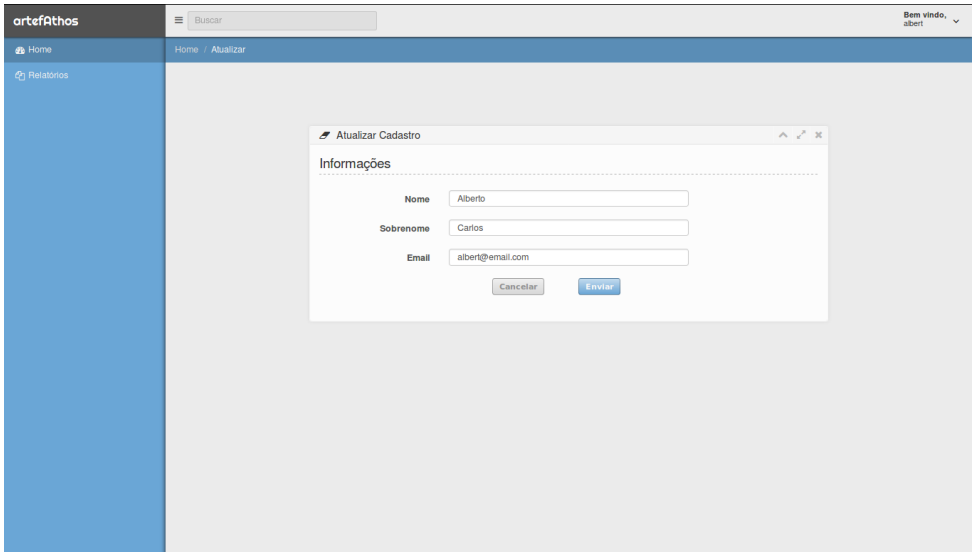

FIG. 6.8: Página do perfil

Existe ainda um a barra de navegabilidade logo abaixo da barra de busca. Sua função é basicamente possibilitar um meio alternativo de navegação entre as páginas que venha a ser mais conveniente conforme o sistema cresça e mais recursos sejam disponibilizados.

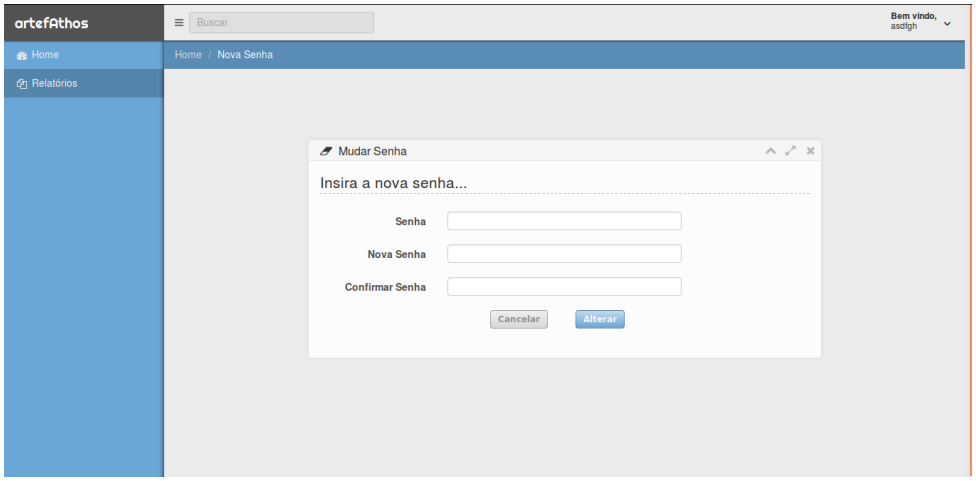

FIG. 6.9: Página de configurações

O sistema está preparado para gerenciar vários usuários e para fazer diversas análises simultaneamente, porém o número de máquinas virtuais utilizadas na análise dinâmica é limitado e definido antes do sistema ser iniciado. Devido a essa limitação das máquinas virtuais, juntamente com o longo tempo necessário para que a análise seja feita, o desem-

penho do sistema pode ser afetado caso a quantidade de análises submetidas seja muito maior que os recursos alocados para fazê-las.

7 CONCLUSÃO

Neste ano de 2017, o malware WannaCry[8] se disseminou muito rapidamente através de técnicas de *phishing*[9] e chegou a contaminar mais de 230.000 sistemas. Este fato acaba sendo mais um fator ratificante da motivação para o desenvolvimento do sistema descrito neste trabalho.

O objetivo inicial do trabalho (desenvolver um sistema capaz de realizar análise de *malware* através das técnicas de varredura por antivírus, análise estática e análise dinâmica) foi completamente atingido. O sistema implementado atende a todos os requisitos necessários, permitindo que o usuário se cadastre, faça *login*, consiga submeter um artefato e acessar os resultados das análises, tudo através de um sistema *web*.

O sistema desenvolvido permite ao usuário realizar a análise de um certo artefato remotamente, sem a necessidade de conhecer a fundo ferramentas de análise, adquirir chaves para utilização de APIs para varredura por antivírus ou a configuração de máquinas virtuais e ambientes controlados. Além disso, é disponibilizada uma interface centralizada e amigável ao cliente, que mantém o histórico de análises anteriores, seus arquivos e resultados.

O *artefAthos* serve muito bem como um protótipo avançado para um possível projeto maior de uma rede de laboratórios virtuais que possam realizar análise de *malwares* a nível nacional. A facilidade advinda de uma estrutura *web* distribuída é de grande valia para qualquer usuário de ambientes acadêmicos ou administrativos com arquivos suspeitos em mãos e pouco ferramental para analisá-los.

Apesar da completude satisfatória do sistema desenvolvido, ainda há espaço para inclusão de novas funcionalidades e integração com outros sistemas, como o FAMa, na forma de entrada para a alimentação de decisores automáticos baseados em Inteligência Artificial. Além de melhoramentos na própria plataforma (implementação da busca por nome, disponibilização de gráficos temporais, etc) há ainda a possibilidade de expansão para novas ferramentas e tipos de análise.

[8]Disponível em: https://pt.wikipedia.org/wiki/WannaCry
[9]Disponível em: https://pt.wikipedia.org/wiki/Phishing

8 REFERÊNCIAS BIBLIOGRÁFICAS

GIANNI AMATO. PEframe 5.0.1. Disponível em: <https://github.com/guelfoweb/peframe>. Acesso em: 3 mai. de 2017.

AV-TEST. Malware. Disponível em: <https://www.av-test.org/en/statistics/malware/>. Acesso em: 3 mar. de 2017.

EPEX. PROJETO ESTRATÉGICO DO EXÉRCITO DE DEFESA CIBERNÉTICA. Disponível em: <http://www.epex.eb.mil.br/index.php/defesa-cibernetica/defesa-cibernetica/escopodciber>. Acesso em: 3 mai. de 2017.

DJANGO SOFTWARE FOUNDATION. Django Overview. Disponível em: <https://www.djangoproject.com/start/overview/>. Acesso em: 19 jul. de 2017.

FUJINO, A.; MURAKAMI, J. ; MORI, T. Discovering similar malware samples using api call topics. In: CONSUMER COMMUNICATIONS AND NETWORKING CONFERENCE (CCNC), 12., 2015. **Proceedings**... Las Vegas, NV, USA: IEEE, 2015, p. 140–147.

MANGIALARDO, R. J. **Integrando as análises estática e dinâmica na identificação de malwares utilizando aprendizado de máquina**. 2015. 134 f. Dissertação (Mestrado em Sistemas e Computação) – Instituto Militar de Engenharia, Rio de Janeiro, 2015. Disponível em: <http://www.comp.ime.eb.br/pos/arquivos/publicacoes/dissertacoes/2015/2015-Reinaldo.pdf>. Acesso em: 3 mai. de 2017.

JAVASCRIPT PARA INICIANTES. O que é Javascript?. Disponível em: <http://tableless.github.io/iniciantes/manual/js/>. Acesso em: 25 jul. de 2017.

SIKORSKI, M.; HONIG, A. **Practical Malware Analysis**. 245 8th Street, San Francisco, CA 94103: No Starch Press, 2012. 766 p.

VIRUSTOTAL. About VirusTotal. Disponível em: <https://virustotal.com/en/about/>. Acesso em: 3 mai. de 2017.

9 APÊNDICES

APÊNDICE 1: ARQUIVO VIEWS.PY DO APP MANAGEUSER

```
from django.shortcuts import render, redirect
from django.http import HttpResponse
from django.contrib.auth import authenticate
from django.contrib.auth import login as auth_login
from django.contrib.auth import logout as auth_logout
from .forms import Login, Register, Update_infos, Change_password
from src.definitions import my_login_required, my_anonymous_required,
    my_create_user, my_update_info_user, my_change_password
from django.contrib.auth.models import User


@my_anonymous_required
def index(request):
    if 'Registrar' in request.POST:
        return register(request)
    else:
        return login(request)


def register(request):
    form_login = Login()
    if request.method == 'POST':
        form_register = Register(request.POST)

        if form_register.is_valid():
            if User.objects.filter(username=form_register.
                cleaned_data['username_register']).exists():
                 form_register.add_error('username_register', 'Usuario
                    ja existe.')
            elif form_register.cleaned_data['password_register'] !=
                form_register.cleaned_data['cpassword']:
```

```
                form_register.add_error('cpassword', 'Senha e
                    confirmacao diferentes')
            else:
                my_create_user(form_register, request)
                return redirect('/')
    else:
        form_register = Register()
    return render(request, 'manageuser/test.html',
                  {'form_register': form_register, 'form_login':
                      form_login, 'headCode': '<title>Cadastro</title>
                      ',
                   'submitValue': 'Registrar'})


def login(request):
    form_register = Register()
    if request.method == 'POST':
        form_login = Login(request.POST)

        if form_login.is_valid():
            user = authenticate(username=form_login.cleaned_data['
                username'],
                                password=form_login.cleaned_data['
                                    password'])
            if user is not None:
                auth_login(request, user)
                return redirect('/')
            else:
                form_login.add_error(None, 'Usuario ou senha
                    incorretos.')
    else:
        form_login = Login()
    return render(request, 'manageuser/test.html',
                  {'form_login': form_login, 'form_register':
                      form_register, 'headCode': '<title>Login</title>
                      ',
                   'submitValue': 'Entrar'})
```

```
@my_login_required
def update_infos(request):
    if request.method == 'POST':
        form = Update_infos(request.POST)
        if form.is_valid():
            my_update_info_user(form, request)
    else:
        form = Update_infos(initial={

            'name': request.user.get_full_name().split()[0],
            'surname': request.user.get_full_name().split()[1],
            'email': request.user.email
        })
    return render(request, 'managefiles/atualizar_cadastro.html',
                  {'form': form, 'headCode': '<title>Atualizar</title
                     >', 'submitValue': 'Atualizar'})


@my_login_required
def change_password(request):
    if request.method == 'POST':
        form = Change_password(request.POST)
        if form.is_valid():
            if not request.user.check_password(form.cleaned_data['
               password']):
                form.add_error('password', 'Senha incorreta')
            elif form.cleaned_data['newPassword'] != form.
               cleaned_data['cnewPassword']:
                form.add_error('cnewPassword', 'Nova senha e
                   confirmacao diferentes')
            else:
                my_change_password(form, request)
    else:
        form = Change_password()
```

```
    return render(request, 'managefiles/nova_senha.html', {'form':
        form, 'headCode': '<title>Alterar senha</title>', 'submitValue
        ': 'Alterar'})


@my_login_required
def logout(request):
    auth_logout(request)
    return redirect('/')
```

APÊNDICE 2: ARQUIVO MODELS.PY DO APP MANAGEUSER

```
from django.db import models
from django.conf import settings
from django.db.models.signals import post_save
import hashlib

class Directories(models.Model):
        user = models.OneToOneField(settings.AUTH_USER_MODEL,
            on_delete=models.CASCADE)
        directory = models.CharField(max_length=32)

        REQUIRED_FIELDS = ['directory']

def create_directory(sender, **kwargs):
        user = kwargs["instance"]

        ho = hashlib.sha256(str(user.id).encode())
        if kwargs["created"]:
                user_directory = Directories(user=user, directory=ho.
                    hexdigest())
                user_directory.save()
post_save.connect(create_directory, sender=settings.AUTH_USER_MODEL)
```

APÊNDICE 3: ARQUIVO FORMS.PY DO APP MANAGEUSER

```
from django import forms

class Login(forms.Form):
        username = forms.CharField(label='Usuario', error_messages={'
            required': 'Usuario nao informado'})
        password = forms.CharField(label='Senha', widget=forms.
            PasswordInput, error_messages={'required': 'Senha nao
            informada'})

class Register(forms.Form):
        name  = forms.CharField(label = 'Nome', required = False)
        surname  = forms.CharField(label = 'Sobrenome', required =
            False)
        username_register = forms.CharField(label='Usuario',
            error_messages={'required': 'Usuario nao informado'})
        password_register = forms.CharField(label='Senha', widget=
            forms.PasswordInput, error_messages={'required': 'Senha
            nao informada'})
        cpassword = forms.CharField(label='Confirmar senha', widget=
            forms.PasswordInput, error_messages={'required': '
            Confirmacao de senha nao informada'})

        email = forms.CharField(label='Email', widget=forms.
            EmailInput, error_messages={'required': 'Email nao
            informado'})

class Update_infos(forms.Form):
        name  = forms.CharField(label = 'Nome', required = False)
        surname  = forms.CharField(label = 'Sobrenome', required =
            False)
```

```
        email = forms.CharField(label='Email', widget=forms.
            EmailInput, error_messages={'required': 'Email nao
            informado'})

class Change_password(forms.Form):
        password = forms.CharField(label='Senha', widget=forms.
            PasswordInput, error_messages={'required': 'Senha nao
            informada'})
        newPassword = forms.CharField(label='Nova senha', widget=
            forms.PasswordInput, error_messages={'required': 'Nova
            senha nao informada'})
        cnewPassword = forms.CharField(label='Confirmar senha',
            widget=forms.PasswordInput, error_messages={'required': '
            Confirmacao de nova senha nao informada'})
```

APÊNDICE 4: ARQUIVO VIEWS.PY DO APP MANAGEFILES

```
from django.shortcuts import render, redirect
from django.views.generic import TemplateView
from django.views.decorators.csrf import csrf_exempt
from django.http import HttpResponse, Http404
from .forms import File_upload, UploadFileForm
from src.definitions import my_login_required, save_uploaded_file
import subprocess
from subprocess import check_output
from pathlib import Path
from distutils.dir_util import copy_tree
import os
import json
from json2html import *
import requests

@my_login_required
def file_upload(request):
        if request.method == 'POST':
                form = File_upload(request.POST, request.FILES)
                if form.is_valid():
                        save_uploaded_file(request.user, request.
                           FILES['file'])
                        return redirect("managefiles:show_directories
                           ")
        else:
                form = File_upload()
        return render(request, 'managefiles/fileform.html', {'form':
           form, 'headCode': '<title>Inicio</title>', 'submitValue':
           'Enviar'})


@my_login_required
```

```
def show_directories(request):
        user = request.user
        reports_directory = check_output(["pwd"]).decode("utf-8")
            [:-1] + "/usr/" + user.directories.directory
        user_directories_list = check_output(["ls", reports_directory
            ]).decode("utf-8").split("\n")
        user_directories_list.pop()
        info_list = []
        for directory in user_directories_list:
            ls = subprocess.Popen(('ls', reports_directory + "/" +
                directory), stdout=subprocess.PIPE)
            output = subprocess.check_output(('grep', '-v', 'reports'
                ), stdin=ls.stdout).decode("utf-8").split("\n")[0]
            ls.wait()
            ready_reports = check_output(["ls", reports_directory + "
                /" + directory + "/reports"]).decode("utf-8").split("\
                n")
            info_list.append({"dir":directory, "file":output, "
                virus_total_ready":"virus_total.json" in ready_reports
                , "static_ready":"static_analysis.json" in
                ready_reports, "dynamic_ready":"
                Arquivos_Analise_Dinamica.zip" in ready_reports})

        return render(request, 'managefiles/relatorios.html', {'
            user_name': user.username, 'info_list': info_list})

@my_login_required
def show_reports(request, report_time):
        return render(request, 'managefiles/reports.html', {'
            user_name': request.user.username, 'report_time':
            report_time})

def download_file(request, report_time, analysis_type, view_method):
        user = request.user
        dir_path = check_output(["pwd"]).decode("utf-8")[:-1] + "/usr
            /" + user.directories.directory + "/" + report_time + "/
            reports"
```

```
        if view_method == "json":
                if (analysis_type == "static_analysis") or (
                    analysis_type == "virus_total"):
                        with open(dir_path + "/" + analysis_type + ".
                            json", "r") as f:
                                response = HttpResponse(f.read(),
                                    content_type="json")
                                response['Content-Disposition'] = '
                                    attachment; filename=' + os.path.
                                    basename(dir_path + "/" +
                                    analysis_type + ".json")
                                return response
                elif analysis_type == "dynamic":
                        with open(dir_path + "/
                            Arquivos_Analise_Dinamica.zip", "r") as f:
                                response = HttpResponse(f.read(),
                                    content_type="zip")
                                response['Content-Disposition'] = '
                                    attachment; filename=' + os.path.
                                    basename(dir_path + "/
                                    Arquivos_Analise_Dinamica.zip")
                                return response
        else:
                with open(dir_path + "/" + analysis_type + ".html", "
                    r") as f:
                     content = f.read()
                return render(request, 'managefiles/reports_html_view
                    .html', {'content': content})

        raise Http404


def make_response(status=200, content_type='text/plain', content=None
    ):
        response = HttpResponse()
        response.status_code = status
```

```
        response['Content-Type'] = content_type
        response.content = content
        redirect("managefiles:show_directories")
        return response

@my_login_required
def home(request):
        return render(request, 'managefiles/forms_file_uploader.html'
            )


class UploadView(TemplateView):
        @csrf_exempt
        def dispatch(self, *args, **kwargs):
                return super(UploadView, self).dispatch(*args, **
                    kwargs)

        def post(self, request, *args, **kwargs):
                form = UploadFileForm(request.POST, request.FILES)
                if form.is_valid():
                        save_uploaded_file(request.user, request.
                            FILES['qqfile'])
                        return make_response(content=json.dumps({ '
                            success': True }))
                else:
                        return make_response(status=400, content=json
                            .dumps({
                                        'success': False,
                                        'error': '%s' % repr(form.
                                            errors)
                                }))
```

APÊNDICE 5: ARQUIVO DEFINITIONS.PY DO DIRETÓRIO SRC

```
from django.shortcuts import redirect
from django.contrib.auth.decorators import login_required
from django.http import QueryDict
from django.contrib.auth.models import User
from django.contrib.auth import authenticate, login
from subprocess import call, check_output, Popen
from datetime import datetime, timezone, timedelta
from time import strftime
import json

def my_login_required(function=None, login_url=None):
        actual_decorator = login_required(function=function,
           redirect_field_name=None, login_url=login_url)
        return actual_decorator

def my_anonymous_required(func):
        def func_wrapper(request):
                if not request.user.is_authenticated():
                        return func(request)
                else:
                        return redirect('/home')
        return func_wrapper


def my_create_user(form, request):
        username = form.cleaned_data['username_register']
        password = form.cleaned_data['password_register']
        email = form.cleaned_data['email']
        name = form.cleaned_data['name']
        surname = form.cleaned_data['surname']

        user = User.objects.create_user(username, email, password)
```

```
        user.first_name = name
        user.last_name = surname
        user.save()

        root_directory = check_output(["pwd"]).decode("utf-8")[:-1]
        user_full_path = root_directory + "/usr/" + user.directories.
            directory
        call(["mkdir", "-p", user_full_path])

        user = authenticate(username=username, password=password)
        if user is not None:
                login(request, user)


def my_update_info_user(form, request):
        request.user.email = form.cleaned_data['email']
        request.user.first_name = form.cleaned_data['name']
        request.user.last_name = form.cleaned_data['surname']
        request.user.save()

def my_change_password(form, request):
        username = request.user.get_username()
        password = form.cleaned_data['newPassword']

        user = User.objects.get(username__exact=request.user.
            get_username())
        user.set_password(form.cleaned_data['newPassword'])
        user.save()

        user = authenticate(username=username, password=password)
        if user is not None:
                login(request, user)

def save_uploaded_file(user, f):
        user_directory = check_output(['pwd']).decode("utf-8")[:-1] +
            "/usr/" + user.directories.directory + "/"
```

```
full_path =  user_directory + datetime.now(tz=timezone(offset
   =timedelta(hours=-3))).strftime("%Y-%m-%d-%H-%M-%S-%f")
call(["mkdir", "-p", full_path])
call(["mkdir", "-p", full_path + "/reports"])
file_path = full_path + "/" + f.name
with open(file_path, 'wb+') as destination:
        for chunk in f.chunks():
                destination.write(chunk)
Popen(["./src/tools.py", full_path, file_path])
```

APÊNDICE 6: COMUNICAÇÃO COM A API DO VIRUSTOTAL

O código abaixo foi utilizado para fazer uso da API do VirusTotal:

```
import requests
import sys
import json

params = {'apikey': '**********'}
files = {'file': (sys.argv[1], open(sys.argv[1], 'rb'))}
response = requests.post('https://www.virustotal.com/vtapi/v2/file/
    scan',
                         files=files, params=params)
json_response = response.json()

params['resource'] = str(json_response['resource'])

headers = {
    "Accept-Encoding": "gzip, deflate",
    "User-Agent" : "gzip, My python requests library example client
        or username",
}

new_response = requests.get('https://www.virustotal.com/vtapi/v2/file
    /report', params=params, headers=headers)
resp = new_response.json()
print(resp)
```

APÊNDICE 7: ARQUIVO TOOLS.PY DO DIRETÓRIO SRC

```
#!/usr/bin/env python3
from distutils.dir_util import copy_tree
from shutil import copyfile
import sys
import os
import json
import requests
import time
import shutil
from json2html import *
from subprocess import call, check_output, run, Popen

def generate_static_reports(full_path, file_path):
    static_analysis_string = check_output(["peframe", "--json",
        file_path]).decode("utf-8")
    static_analysis_dict = json.loads(static_analysis_string)
    strings_report_string = check_output(["peframe", "--strings",
        file_path]).decode("utf-8")
    strings_report_dict = json.loads(strings_report_string)
    static_analysis_dict.update(strings_report_dict)
    final_string = json.dumps(static_analysis_dict, indent=4,
        sort_keys=False)
    with open(full_path + "/reports/static_analysis.json", "w+") as
        outputfile:
            outputfile.write(final_string)

    with open(full_path + "/reports/static_analysis.html", "w+") as
        outputfile:
         outputfile.write(json2html.convert(json =
            static_analysis_dict))

def generate_dynamic_reports(full_path, file_path):
```

```
        REST_URL = "http://localhost:8090/tasks/create/file"
        FILE = file_path

        with open(FILE, "rb") as sample:
                files = {"file": ("temp_file_name", sample)}
                r = requests.post(REST_URL,  files=files)

        task_id = r.json()["task_id"]

        status = 'not reported'

        while status != 'reported':
                r = requests.get("http://localhost:8090/tasks/view/"
                   + str(task_id))
                status = r.json()["task"]["status"]
                time.sleep(60)
        os.chdir(full_path + "/reports")
        path_analyses_cuckoo = "/home/artefathos/.cuckoo/storage/
           analyses/" + str(task_id)
        shutil.make_archive("Arquivos_Analise_Dinamica", "zip",
           path_analyses_cuckoo)

        copyfile(path_analyses_cuckoo + "/reports/report.html",
           full_path + "/dynamic_analysis.html")


def generate_virus_total_reports(full_path, file_path):
        run(["./src/queue_tools/VTLockCheck", "src/queue_tools/
           virusTotalLock"])
        virus_total_string = json.dumps(virusTotal(file_path), indent
           =4, sort_keys=False)
        with open(full_path + "/reports/virus_total.json", "w+") as
           outputfile:
                outputfile.write(virus_total_string)

        virus_total_dict = json.loads(virus_total_string)
```

```
        with open(full_path + "/reports/virus_total.html", "w+") as
            outputfile:
             outputfile.write(json2html.convert(json =
                virus_total_dict))

def virusTotal(path):
    params = {'apikey': '95237
        e1de590ecf93b71c02679cd6ba797497f8d4aa3bd5483f2b51bb4015708'}
    files = {'file': (path, open(path, 'rb'))}
    response = requests.post('https://www.virustotal.com/vtapi/v2/
        file/scan', files=files, params=params)
    json_response = response.json()


    params['resource'] = str(json_response['resource'])

    headers = {
        "Accept-Encoding": "gzip, deflate",
        "User-Agent" : "gzip, My python requests library example
            client or username",
    }

    new_response = requests.get('https://www.virustotal.com/vtapi/v2/
        file/report', params=params, headers=headers)
    resp = new_response.json()
    return resp

full_path = sys.argv[1]
file_path = sys.argv[2]
generate_static_reports(full_path, file_path)
generate_virus_total_reports(full_path, file_path)
generate_dynamic_reports(full_path, file_path)
```

10 ANEXOS

ANEXO 1: CATÁLOGO DE APIS

- CreateFile: cria um arquivo ou abre um arquivo existente. Pode ser utilizado por um *malware* para despejar um arquivo no sistema.
- CreateMutex: cria uma exclusão mútua de objeto que pode ser usada por um *malware*.
- CreateProcess: cria e inicia um novo processo.
- CreateRemoteThread: utilizado para iniciar uma *thread* em um processo remoto. Um *malware* pode utilizá-lo para injetar código em um processo existente.
- CreateService: cria um serviço que pode ser iniciado em tempo de *boot*. Um *malware* pode utilizá-lo para persistência ou para carregar *drivers*.
- OpenFile: abre um arquivo.
- DeleteFile: exclui um arquivo.
- FindWindow: busca por uma janela aberta no *Desktop*.
- OpenMutex: abre um *handle* para um objeto com exclusão mútua. Um *malware* pode utilizar uma exclusão mútua para evitar a infecção de um sistema por diferentes instâncias de um mesmo *malware*. Exemplo: quando um *trojan* infecta um ambiente o primeiro passo é obter um *handle* para um mutex nomeado, se o processo falhar o processo do *malware* é encerrado.
- OpenSCManager: abre um *handle* para o gerenciador de controle de serviços. Isto permitirá ao *malware* interagir com os processos dos serviços do Windows, iniciando-os ou parando-os.
- ReadFile: lê um arquivo.
- ReadProcessMemory: usado para ler a memória de um processo remoto. Regiões válidas de memória podem ser copiadas utilizando esta API. Um *malware* pode fazer uso desta API para obter, por exemplo, o número do cartão de crédito de um usuário, utilizando um código de busca.

- RegDeleteKey: apaga uma chave de registro e subchaves.
- RegEnumKeyEx: enumera as subchaves de um registro aberto. Pode ser utilizado em conjunto com RegDeleteKey para apagar recursivamente chaves de registro.
- RegEnumValue: enumera os valores para a chave de registro aberta.
- CreateSection: cria um objeto de seção.
- RegOpenKey: abre um *handle* para controlar a leitura e edição de um registro.
- ShellExecute: utilizado para executar um outro programa.
- TerminateProcess: termina um processo e todas as suas threads.
- URLDownloadToFile: faz um *download* de n bits da Internet e os salva em um arquivo.
- WriteFile: grava os dados no dispositivo de saída.
- WriteProcessMemory: grava dados em um processo remoto. Utilizado na injeção de código.
- ZwMapViewOfSection: mapeia a visão de uma seção em um espaço de endereçamento virtual.
- LoadDll: função de baixo nível que carrega uma DLL em um processo. Pode indicar que o programa atua de forma furtiva.
- GetProcedureAddress: recupera o endereço de uma função carregada na memória. Usada para importar funções de outras DLLs em adição às importadas no cabeçalho do arquivo PE.
- OpenKey: abre um *handle* para controlar a leitura e edição de um registro.
- QueryValueKey: acessa os valores de uma chave de registro.
- IsDebuggerPresent: verifica se o processo está sendo depurado. Utilizado na detecção de *debuggers*.
- GetSystemMetrics: obtém informações de configurações do sistema.
- SetInformationFile: altera as informações de um arquivo.

- CreateMutant: um objeto mutante é criado e é aberto um *handle* para ele.
- OpenSection: abre um *handle* para uma seção.
- SetWindowsHookExA: define uma *hook function* que será chamada quando um evento ocorrer. Normalmente é utilizada por *keyloggers* e *spywares*.
- RegQueryValueEx: retorna o tipo e o valor para o nome específico associado a uma chave de registro.
- RegCloseKey: fecha o *handle* de uma chave de registro.
- OpenMutant: abre um *handle* para o objeto para execução exclusiva da instância do *malware*.
- LdrGetDllHandle: obtém o *handle* de um objeto.
- FreeVirtualMemory: libera uma região de páginas dentro do espaço de endereços virtuais de memória de um processo.