UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

ANÁLISE DE OBJETOS A PARTIR DA EXTRAÇÃO DA MEMÓRIA RAM DE SISTEMAS SOBRE ANDROID RUN-TIME (ART)

ALBERTO MAGNO MUNIZ SOARES

ORIENTADOR: RAFAEL TIMÓTEO DE SOUSA JR.

DISSERTAÇÃO DE MESTRADO EM ENGENHARIA ELÉTRICA
ÁREA DE CONCENTRAÇÃO INFORMÁTICA FORENSE E SEGURANÇA DA INFORMAÇÃO

PUBLICAÇÃO: PPGENE.DM - 628/2016
BRASÍLIA / DF: DEZEMBRO/2016

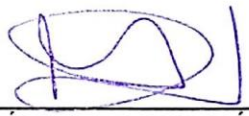
UNIVERSIDADE DE BRASÍLIA
FACULDADE DE TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA

ANÁLISE DE OBJETOS A PARTIR DA EXTRAÇÃO DA MEMÓRIA RAM DE SISTEMAS SOBRE ANDROID RUN-TIME (ART)

ALBERTO MAGNO MUNIZ SOARES

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

APROVADA POR:



RAFAEL TIMÓTEO DE SOUSA JÚNIOR, Dr., ENE/UNB
(ORIENTADOR)

WILLIAM FERREIRA GIOZZA, Dr., ENE/UNB
(EXAMINADOR INTERNO)

BRUNO WERNECK PINTO HOELZ, Polícia Federal
(EXAMINADOR EXTERNO)

Brasília, 16 de Dezembro de 2016.

FICHA CATALOGRÁFICA

SOARES, ALBERTO MAGNO MUNIZ SOARES
ANÁLISE DE OBJETOS A PARTIR DA EXTRAÇÃO DA MEMÓRIA RAM DE SISTEMAS SOBRE ANDROID RUN-TIME (ART) [Distrito Federal] 2016.
xiv, 116p., 297 mm (ENE/FT/UnB, Mestre, Engenharia Elétrica, 2016).

Dissertação de Mestrado – Universidade de Brasília, Faculdade de Tecnologia. Departamento de Engenharia Elétrica.

1. Memória Volátil 2. Computação Forense
3. Android

I. ENE/FT/UnB. II. Título (Série)

REFERÊNCIA BIBLIOGRÁFICA

SOARES, A. M. M. (2016). Análise de objetos a partir da extração da memória RAM de sistemas sobre Android Run-Time (ART). Dissertação de Mestrado, Publicação PPGENE.DM - 628/2016, Departamento de Engenharia Elétrica, Universidade de Brasília, Brasília, DF, 116p.

CESSÃO DE DIREITOS

NOME DO AUTOR: Alberto Magno Muniz Soares
TÍTULO DA DISSERTAÇÃO: Análise de Objetos a partir da Extração da Memória RAM de Sistemas sobre Android Run-Time (ART).
GRAU/ANO: Mestre/2016.

Alberto Magno Muniz Soares
SPI/IC/PCDF – SPO, Conjunto A, Lote 23
CEP 70610-200 – Brasília – DF - Brasil

A Deus pela vida e seu amor manifestado na minha família, especialmente no carinho da minha esposa maravilhosa, na alegria dos meus filhos, na dedicação e paciência infindável dos meus pais, na lealdade dos meus irmãos, no apoio infalível da minha sogra, no bem querer e tolerância dos meus amigos e colegas, e em todos que me motivaram a seguir adiante no caminho do bem.

AGRADECIMENTOS

Ao meu orientador Prof. Dr. Rafael Timóteo de Sousa Júnior, pelo apoio e dedicação essenciais para o desenvolvimento deste trabalho e para o meu desenvolvimento como pesquisador.

Aos colegas do Instituto de Criminalística do Distrito Federal, pelas conversas enriquecedoras, dedicação ao trabalho, colaboração e amizade.

Aos profissionais do Laboratório de Redes (LabRedes) do Departamento de Engenharia Elétrica da Universidade de Brasília, pela cordialidade habitual, especialmente ao Luciano Pereira Anjos.

A todos, os meus sinceros agradecimentos.

O presente trabalho foi realizado com o apoio do Instituto de Criminalística do Distrito Federal – IC/PCDF e do Departamento Polícia Federal – DPF, com recursos do Programa Nacional de Segurança Pública com Cidadania – PRONASCI, do Ministério da Justiça.

RESUMO

ANÁLISE DE OBJETOS A PARTIR DA EXTRAÇÃO DA MEMÓRIA RAM DE SISTEMAS SOBRE ANDROID RUN-TIME (ART)

Autor: Alberto Magno Muniz Soares

Orientador: Rafael Timóteo de Sousa Jr.

Programa de Pós-graduação em Engenharia Elétrica

Brasília, dezembro de 2016

Este trabalho tem o objetivo de apresentar uma técnica de análise de objetos em memória no ambiente de execução ART (Android Run-Time) a partir de uma extração de dados da memória volátil. Um estudo do código fonte AOSP (Android Open Source Project) foi feito para entendimento do ambiente de execução utilizado no sistema operacional Android moderno, e foram elaboradas ferramentas de software que permitem a localização, extração e interpretação de dados úteis para o contexto forense. Construídas como extensões para o *framework* Volatility, essas ferramentas possibilitam localizar, em uma extração de memória de um dispositivo com arquitetura ARM, instâncias de classes arbitrárias e suas propriedades de dados.

ABSTRACT

OBJECTS ANALISYS BASED ON RAM MEMORY EXTRACTION OVER ANDROID RUN-TIME (ART) SYSTEMS

Author: Alberto Magno Muniz Soares

Supervisor: Rafael Timóteo de Sousa Jr.

Programa de Pós-graduação em Engenharia Elétrica

Brasília, december of 2016

The work in this thesis aims at describe a technique for analyzing objects in memory within the execution environment ART (Android Run-Time) from a volatile memory data extraction. A study of the AOSP (Android Open Source Project) source code was necessary to understand the runtime environment used in the modern Android operating system, and software tools were developed allowing the location, extraction and interpretation of useful data for the forensic context. Built as extensions for the Volatility Framework, these tools enable to locate, in a memory extraction from a device with ARM architecture, arbitrary instances of classes and their data properties.

SUMÁRIO

LISTA DE TABELAS

LISTA DE FIGURAS

LISTA DE SÍMBOLOS, NOMENCLATURA E ABREVIAÇÕES

ADB	**Android Debug Bridge**
AOSP	**Android Open Source Project**
AOT	**Ahead-Of-Time**
API	**Aplication Program Interface**
APK	**Android Package**
ARM	**Advanced RISC Machine**
ART	**Android Run-Time**
ASHMEM	**Anonymous Shared Memory**
ASL	**Apache License**
ASLR	**Address Space Layout Randomization**
CMS	**Concurrent Mark Sweep**
COM	**Component Object Model**
CORBA	**Common Object Broker Request Architecture**
DEX	**Dalvik EXecutables**
DMD	**Droid Memory Dumper**
DVM	**Dalvik Virtual Machine**
ELF	**Executable and Linking Format**
FDE	**Full Disk Encryption**
FROST	**Forensic Recovery Of Scrambled Telephones**
GC	**Garbage Collection**
GNU GPL	**GNU General Public License**
IPC	**Inter-Process Comunication**
JDWP	**Java Debug Wire Protocol**
JIT	**Just-in-Time**
JNI	**Java Native Interface**
JPG	**Joint Photographic Experts Group**
JTAG	**Joint Test Action Group**
JVM	**Java Virtual Machine**
LiME	**Linux Memory Extrator**
LLVM	**Low Level Virtual Machine**
LOS	**Large Object Space**
ODEX	**Optimized DEX**
PDF	**Portable Document Format**
PIC	**Position-Independent Code**
RAM	**Random Access Memory**
RosAlloc	**Runs-of-Slots-Allocator**
TCP	**Transmission Control Protocol**
TLA	**Thread Local Allocation**
VDSO	**Virtual Dynamic Shared Object**

1. INTRODUÇÃO

Os dispositivos móveis pessoais podem ser utilizados para muitas finalidades e sua memória volátil pode conter potenciais evidências digitais para uma investigação.

Tradicionalmente, os exames em dispositivos móveis são focados na aquisição e análise de dados presentes nas mídias de armazenamento não voláteis, pois, em função do objetivo da investigação ou devido a dificuldade com relação à natureza efêmera dos dados, análises da memória volátil não são comumente realizadas. Mas, com o uso crescente da criptografia e a presença de software malicioso, cada vez mais sofisticados, a necessidade de conduzir uma investigação sobre o conteúdo da memória volátil de dispositivos móveis tem ficado em forte evidência. A comunidade forense, conforme [BREZINSKI e KILLALEA 2002], observa a priorização dos dados da memória na ordem de volatilidade das evidências digitais, uma vez que algumas informações sobre o ambiente do sistema nunca são mantidas estaticamente em mídia de armazenamento secundário. Assim, torna-se importante a existência de técnicas que permitam a análise de dados de uma extração de memória volátil de forma mais aprofundada que as técnicas tradicionais.

O Android é um sistema operacional baseado no kernel Linux e projetado principalmente para dispositivos móveis. Esse sistema já contabilizou, segundo [STAMFORD 2014], mais de um bilhão de usuários até 2014, liderando o mercado de sistemas operacionais, nas versões de 32-bits e 64-bits, de processadores x86, MIPS e, principalmente, o ARM.

Para as subsequentes versões desse sistema operacional, foram desenvolvidos estudos, tais como [SIMÃO et al. 2011] e [SYLVE et al. 2012], sobre a extração e a análise de memória específicas dos dispositivos portadores, pois, apesar de poder ser considerado uma distribuição Linux, o Android possui particularidades que exigem o entendimento detalhado do ambiente de execução e o uso de técnicas especializadas.

Ao contrário de outras plataformas móveis como iOS, Windows Phone ou Tizen, que executam código previamente compilado especificamente para suas arquiteturas de hardware, a maioria do software que é executado nas plataformas Android é baseada em um código genérico para uma máquina virtual denominada *Dalvik Virtual Machine* (DVM). O código para essa máquina, o *bytecode,* é compilado ou interpretado no próprio dispositivo em instruções nativas, permitindo que uma mesma aplicação possa ser distribuída para

diversas arquiteturas de processadores. Dessa forma, a análise de dados da memória RAM de uma aplicação Android necessita de técnicas específicas que permitam ir além das técnicas tradicionais de recuperação aplicadas sobre ambiente Linux, alcançando a recuperação de informações sobre os objetos da máquina virtual, administrados pelo ambiente de execução.

Na versão 4.4, denominada KitKat, lançada em outubro de 2013, como apresentado em [GOOGLE 2014], um novo ambiente de execução *Android Run-Time* (ART) foi disponibilizado, numa versão "beta", em substituição ao ambiente de execução e interpretação da DVM, operando com uma nova estratégia de execução consoante ao potencial do hardware disponível. A partir da versão 5.0, lançada em novembro de 2014, denominada Lollipop, a compilação baseada na estratégia *Just-in-Time* (JIT) foi definitivamente substituída por uma compilação de toda aplicação durante a sua instalação, denominada *Ahead-Of-Time* (AOT), e o ambiente de execução passou a ter novos mecanismos de gerenciamento de memória visando maior eficiência. Conforme publicado no site do projeto *Android Open Source Project* (AOSP), em agosto de 2016, as versões com essa mudança, já representam grande parte dos dispositivos Android disponíveis no mercado, conforme Tabela 1.1 e Figura 1.1.

Tabela 1.1 - Distribuição da plataforma Android por versão

Versão	Apelido	Distribuição
2.2	Froyo	0,1%
2.3.3 - 2.3.7	Gingerbread	1,7%
4.0.3 - 4.0.4	Ice Cream Sandwich	1,6%
4.1.x	Jelly Bean	6,0%
4.2.x		8,3%
4.3		2,4%
4.4	**KitKat**	**29,2%**
5.0	**Lollipop**	**14,1%**
5.1		**21,4%**
6.0	**Marshmallow**	**15,2%**

Fonte: Retirada de [GOOGLE 2016]

Fonte: Retirada de [GOOGLE 2016]

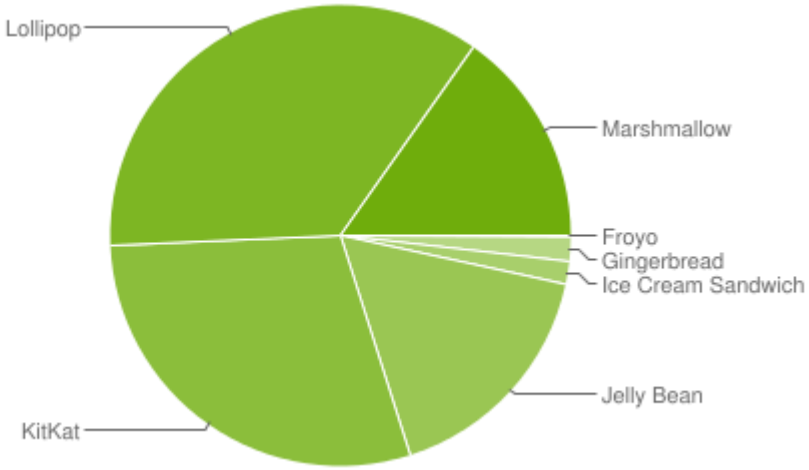


Figura 1.1 - Gráfico sobre a distribuição da plataforma por versão.

O código fonte do Android é liberado sobre licença de software livre *Apache License 2.0* (ASL) - pelo projeto AOSP, e o *kernel*, sob a licença *GNU General Public License* (GNU GPL). A possibilidade de ter acesso aos códigos fontes, incluindo a implementação do ambiente de execução, é essencial para o entendimento do ambiente de execução, pois permite um conhecimento aprofundado de como se organizam na memória as estruturas de dados de um aplicativo.

Por meio de algumas técnicas de aquisição de memória volátil disponíveis, em [APOSTOLOPOULOS et al. 2013] foi apresentado estudo sobre recuperação de credenciais de aplicações em dispositivos Android, sendo demonstrado, sem a análise dos objetos das aplicações, que credenciais de aplicações estão acessíveis por inspeção direta dos dados extraídos. Nas análises dos dados provenientes, tanto dos dispositivos reais como dos emulados, é mencionado que não houve grandes discrepâncias nos resultados.

Em [WÄCHTER e GRUHN 2015] foi analisada a praticidade das técnicas de extração de memória RAM de dispositivos Android para fins forenses, explicitando limitações condicionadas a características intrínsecas implementadas pelos fabricantes, como mecanismos de segurança rígidos que impedem o acesso aos dados. Dentre as diferentes técnicas analisadas, é bem difundida a desenvolvida em [SYLVE et al. 2012] onde foi apresentada a técnica de aquisição de dados chamada *Droid Memory Dumper* (DMD), que posteriormente foi renomeada para *Linux Memory Extrator* (LiME). Essa

técnica possui alta taxa de integridade dos dados originais, permitindo obter dados da memória volátil de um dispositivo "rooteado"[1]. Mas, a técnica exige o carregamento de um módulo *kernel* no dispositivo alvo. Este módulo *kernel* opera copiando os dados da memória RAM, observando o mapeamento dos espaços de endereçamento provido pela estrutura `iomem_resource` do *kernel*, permitindo o correto mapeamento de cada página de memória dos endereços físicos para os virtuais, e possibilitando gravar esses dados em cartão de memória ou transmiti-los através de um socket TCP. Também em [Sylve et al. 2012] foram desenvolvidas ferramentas para o *framework* Volatility para análise de estruturas de memória do kernel Linux Android.

Como um contorno às limitações no processo de extração, em [HILGERS et al. 2014] foi apresentado um trabalho de recuperação de dados baseado na aquisição de dados de dispositivos reais com a versão do Android inferior a 4.4, utilizando a técnica *Forensic Recovery Of Scrambled Telephones* - FROST. O trabalho afirma que mesmo com a reinicialização e o apagamento irrecuperável dos dados da memória não volátil, que ocorre quando da restauração para configuração de fábrica provocada pelo desbloqueio do *bootloader*[2] de alguns aparelhos, ainda assim é possível localizar e recuperar dados de interesse forense remanescentes na RAM, utilizando ferramentas compostas de extensões específicas para o *framework* Volatility, mas limitadas a recuperação de objetos nos ambientes de execução anteriores ao ART. O *framework* Volatility é um conjunto de ferramentas abertas que foram utilizadas nesse trabalho, implementadas em Python, sob a licença GNU-GPL, para extração de artefatos digitais de amostras de memória volátil, e foi concebido para funcionar em qualquer plataforma de sistema operacional, para a qual exista implementação de Python, de forma totalmente independente do sistema em investigação [VOLATILESYSTEMS 2015].

Em [BACKES et al. 2016] foi apresentado uma solução de instrumentação de aplicações baseada no processo de compilação disponível no ART, ressaltando as inovações no processo de compilação, incluindo detalhes de funcionamento interno importantes para compreensão do ambiente de execução ART e suas diferenças com relação aos anteriores.

---

[1] O processo de ganho de privilégios no Android é denominado "rooting" ou "rooteamento", uma derivação do termo root, que define o nome da conta de super usuário do Linux. Essa conta tem direitos e permissões sobre todos os arquivos e programas em um sistema, possibilitando controle total sobre o sistema operacional

[2] Programa executado antes do carregamento do sistema operacional para inicialização do sistema.

Até o momento, a quantidade de trabalhos abordando o ambiente de execução em Android nas versões iguais ou superiores a 5.0 é escassa, sendo o estudo do código do AOSP e de suas constantes atualizações uma importante fonte de informação, também não se conhece técnica forense específica para análise de memória volátil para ambiente de execução da plataforma Android nas versões superiores a 5.0. Assim, o presente trabalho trata de um estudo do ambiente de execução ART e de uma técnica de análise de objetos Java, a partir da extração de memória volátil de um ambiente de execução Android 5.0, baseado no código fonte disponibilizado pelo AOSP, para permitir a localização e a recuperação estruturada de dados de objetos de uma aplicação em execução.

1.1. ÁREAS DE APLICAÇÃO DOS RESULTADOS

Segundo [CARRIER 2003], o exame de evidências digitais pode ser decomposto em cinco tarefas: recuperação de dados, análise de dados, extração de evidências e preservação, e apresentação de evidências. Este trabalho visa contribuir com uma técnica para tarefa de análise dos dados.

Conforme [LIGH et al. 2014], a análise de memória volátil de dispositivos móveis possibilita uma visão do estado de execução de um sistema, como quais processos estão em execução, conexões de rede ativas, ou comandos recentemente executados. Existem dados críticos que estão exclusivamente na memória volátil, como chaves de encriptação, fragmentos de código injetados, mensagens de aplicativos de conversa não persistentes, mensagens de e-mail decifradas ou registros do histórico de navegação que não foram armazenados. Na detecção de códigos maliciosos onde se utilizam de técnicas arrojadas, a análise da memória volátil pode ser uma estratégia importante na detecção e entendimento do funcionamento de um *malware*. Inclusive existem técnicas, como as apresentadas em [VIDAS E CHRISTIAN 2014] e [PETSAS 2014], onde *malware* se desativam quando são executados em ambientes emulados, o que dificulta a detecção por técnicas de análise dinâmica tradicionais.

Para as versões da plataforma Android a partir da versão 5.0 ainda existem poucos trabalhos que tratam das estruturas contidas no ambiente de execução dessas versões, e o presente trabalho avança no sentido de entender as estruturas de memória envolvidas na execução de uma aplicação e no desenvolvimento de uma técnica que permita a análise de dados extraídos da memória volátil com enfoque forense, pois, sem uma análise

aprofundada das estruturas de memória envolvidas no funcionamento do sistema operacional e do ambiente de execução, a análise é limitada a métodos de *carving* como os utilizados na recuperação de arquivos do tipo JPEG ou PDF (por exemplo, utilizando as ferramentas Foremost, Scalpel ou Photorec), sequências de texto abertos, e outros artefatos que independem do entendimento das estruturas de armazenamento em memória, limitada a localização por padrões intrínsecos aos artefatos.

Com a técnica proposta, é possível efetuar uma análise estruturada dos objetos Java de uma aplicação em execução que auxilie procedimentos como análise de *malware*, engenharia reversa e recuperação de dados.

1.2. HIPÓTESE DE TRABALHO E METODOLOGIA

A hipótese original deste trabalho foi alterada no decorrer de seu desenvolvimento. Inicialmente, o objetivo da pesquisa era definir uma técnica de detecção de *malware* baseada na análise dos dados coletados a partir da extração de memória volátil com uso de informações coletadas do ambiente de execução de uma aplicação. No entanto, foi constatado durante o estudo que a arquitetura interna do sistema operacional Android havia sofrido alterações a partir da versão 4.4 e que os trabalhos disponíveis sobre o novo ambiente de execução eram escassos, sendo necessário aprofundar o conhecimento das novas estruturas internas envolvidas para poder avançar na hipótese original.

Com isso, a hipótese deste trabalho foi alterada para a possibilidade de se definir uma técnica de análise de memória volátil de dispositivos com ambiente de execução ART, baseado na análise do código fonte e na documentação disponível, que permita a partir de uma aquisição de memória volátil, definir a localização e realizar a recuperação de dados de objetos Java de um aplicativo em execução.

Para isso, foi feito estudo do código fonte da versão 5.0.1_r1 do sistema operacional, disponível no site do AOSP, e pesquisa em trabalhos anteriores, com enfoque nas estruturas relacionadas com o ambiente de execução, além da elaboração de técnica de recuperação de dados de objetos de uma determinada aplicação a partir de uma extração total de memória.

Para validação da técnica, foi realizada análise de dados adquiridos da memória volátil de um dispositivo emulado e de outro real.

1.3. OBJETIVOS

Se espera desse trabalho o entendimento suficiente das estruturas de memória envolvidas no processo de execução de aplicações Android sobre o ART e com isso se construir uma técnica de análise de dados de extração de memória RAM eficaz, capaz de localizar e analisar as estruturas do ambiente de execução, permitindo recuperar dados de objetos Java de uma aplicação alvo em execução.

Com esse entendimento e uma técnica eficaz é possível uma análise mais aprofundada do que as técnicas tradicionais em procedimentos relacionados com análise de *malware*, engenharia reversa e recuperação de dados.

1.4. ESTRUTURA DESSE TRABALHO

O conteúdo dessa dissertação está estruturado da seguinte forma:

Capítulo dois introduz os conceitos do sistema operacional Android que são essencias para o entendimento do trabalho. É feita uma apresentação da arquitetura do sistema delineando aspectos referentes ao gerenciamento de memória pelo ambiente de execução.

Capítulo três apresenta a técnica de análise de objetos de uma aplicação Android. Apresenta-se os procedimentos de aquisição, extração e análise de dados de uma aplicação, incluindo ferramentas construídas para extração e decodificação de objetos Java gerenciados pelo ambiente de execução.

O capítulo quatro apresenta a validação experimental da técnica de análise utilizando as ferramentas construídas operando sobre dados de memória volátil adquiridos de um aparelho emulado e de um aparelho real.

Finalmente, o capítulo cinco traz um sumário do trabalho desenvolvido na dissertação, conclusões, limitações, e possíveis trabalhos futuros.

2. CONCEITOS DA ARQUITETURA ANDROID

2.1. VISÃO GERAL

A plataforma Android é constituída de uma pilha de software com três camadas principais: uma de aplicações, uma contendo um *framework* de objetos (*framework Android)* e o ambiente de execução (*Runtime* – RT) para execução de *bytecode* de máquina virtual (*Virtual Machine* – VM), e uma de código nativo contendo *kernel* Linux com bibliotecas de abstração de hardware [YAGHMOUR 2013].

Fonte: Adaptação de figura em [YAGHMOUR 2013]

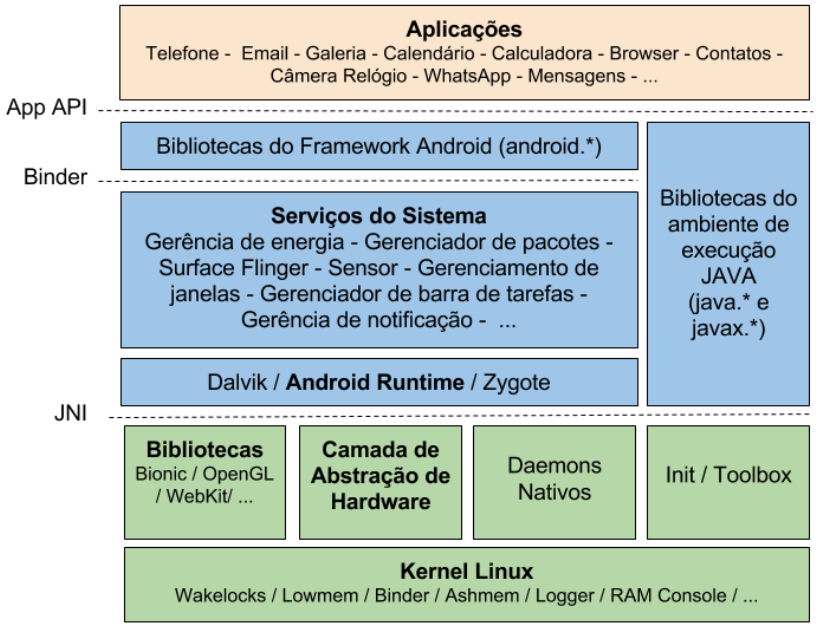


Figura 2.1 - Camadas da arquitetura Android

Como ilustrado na Figura 2.1, na camada superior é oferecida a possibilidade de se estender e incrementar o funcionamento dos dispositivos com aplicativos que abstraem as camadas inferiores. Na camada intermediária que contém o *framework* Android é oferecida uma *Aplication Program Interface* (API) com serviços para as aplicações, como mecanismo de inter-comunicação de aplicações, gerenciamento de interface do usuário, acesso a repositórios de dados. Alguns desses serviços se comunicam com outros serviços em nível de *kernel* e drivers de dispositivos, enquanto outros funcionam como facilitadores para operações em baixo nível. Na camada de código nativo, está presente o *kernel* Linux com os serviços de sistema, serviços de rede, e bibliotecas de uso geral.

2.2. SISTEMA OPERACIONAL LINUX

O Android foi projetado para ser compatível com vários tipos de hardwares. Isso é possível, em grande parte, por meio do uso do *kernel* Linux, presente em uma grande variedade de hardware, em conjunto com um ambiente para execução de *bytecode* de máquina virtual.

Segundo [CASTRO 2016], o Android é considerado, pela Linux Foundation[3], uma distribuição Linux, inicialmente baseada na versão 2.6.25, com modificações peculiares. Segundo [YAGHMOUR 2013], algumas modificações estão relacionadas com sistema de gerenciamento de energia, gerenciamento de memória, comunicação inter-processos, e memória compartilhada. Atualmente, a maior parte das versões do Android utilizam a versão 3.4 ou superior [DRAKE et al. 2014].

Nos sistemas Linux, para haver a comunicação entre processos, geralmente são utilizados os denominados mecanismos de *Inter-Process Comunication* (IPC) como arquivos, sinais, sockets, pipes, semáforos, memória compartilhada, filas de mensagens, e outros. No Android são utilizados alguns desses mecanismos (p. ex. sockets locais), mas não semáforos, segmentos de memória compartilhados e filas de mensagens. Um dos mecanismos IPC utilizado é o mecanismo denominado *Binder*, semelhante a outros sistemas de IPC, mas sem chamadas remotas [ELENKOV 2014]. Um dos usos desse mecanismo é permitir aos processos identificar de forma segura e controlada os descritores de regiões de memória alocados.

Com o uso do *kernel* Linux, o processo de análise de uma extração de memória volátil é facilitado por contar com o conhecimento disponível das estruturas de memória utilizadas pelo *kernel* do sistema, o que possibilita, através de ferramentas específicas e com pequenas adaptações, a recuperação de vários tipos de dados, como lista de processos em execução, lista de descritores de arquivos, mapeamento de memória incluindo informações das regiões de *heap*, pilha, bibliotecas compartilhadas, e outros mapeamentos de memória em arquivo.

[3] A Linux Foundation (LF) é uma organização sem fins lucrativos que fomenta o crescimento do Linux com membros formados pelas principais empresas de Linux e Open Source e desenvolvedores de todo o mundo.

Algumas dessas informações obtidas do *kernel*, como lista de processos e mapeamentos de memória, são necessárias no processo de recuperação de objetos do ambiente de execução desenvolvido nesse trabalho.

2.3. MÁQUINA VIRTUAL DALVIK - DVM

A máquina virtual do Android, *Dalvik Virtual Machine* (DVM), é semelhante a uma *Java Virtual Machine* (JVM). Ela pode ser classificada como uma máquina virtual de aplicação, onde a máquina virtual é destinada a surportar apenas um processo ou aplicação convidada específica promovendo uma virtualização de uma linguagem de programação Java [LAUREANO E MAZIERO 2008]. Dessa forma, uma aplicação é codificada em uma representação intermediária para essa máquina, conhecida pelo termo *bytecode*, que pode ser transportada para várias arquiteturas distintas. Como será detalhado em item referente ao processo de inicialização do Android, cada aplicação em execução está associada a seu próprio ambiente de execução de máquina virtual.

Uma DVM possui vários mecanismos em comum com uma JVM, inclusive o sistema de gerenciamento automático de memória conhecido como *Garbage Collection* – GC. Em essência, essas máquinas se diferenciam basicamente no conjunto de instruções e no formato dos arquivos que contém o *bytecode* das classes.

Como ilustrado na Figura 2.2, os arquivos contendo código para DVM, denominados DEX - *Dalvik EXecutables*[4], se diferem basicamente dos arquivos de código de uma JVM pelo fato de que em um único arquivo estão contidas todas as classes, enquanto que na JVM as classes estão espalhadas por diversos arquivos. Isso traz a possibilidade de maior compressão dos arquivos DEX visando o uso em dispositivos com limitações de memória e processamento.

O empacotamento das aplicações contendo *bytecode*, recursos e bibliotecas de código nativo, são armazenados em um arquivo compactado denominado *Android Package* (APK).

[4] Leiaute do arquivo contendo informações das classes e dados associados está descrito no site do projeto Android em `https://source.android.com/devices/tech/dalvik/dex-format.html`

Fonte: Adaptação de figura em [BECHTSOUDIS 2015]

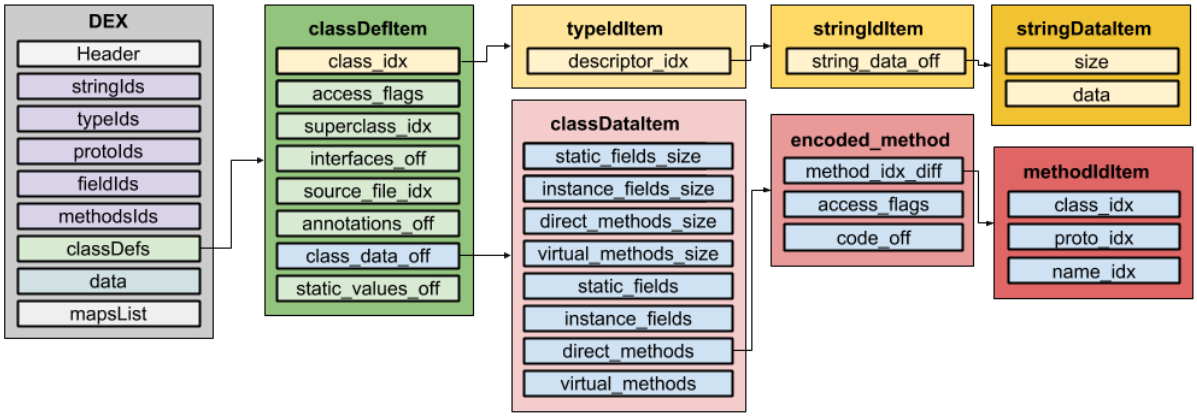


Figura 2.2 - Estrutura parcial de um arquivo DEX

2.4. AMBIENTE DE EXECUÇÃO

O ambiente de execução é responsável por gerenciar os aplicativos Android destinados a operar sobre o *framework* Android. Dentre suas responsabilidades estão as de fornecer o gerenciamento de memória da DVM e o acesso a outros serviços do sistema como compilação/interpretação, mapeamento de memória e carregamento de código.

A partir da versão 5.0 do Android, foi disponibilizado um novo ambiente de execução denominado *Android Run-Time* (ART), com compilação para código nativo anterior à execução, em substituição ao ambiente DALVIK (mesma denominação da máquina virtual) com sistema de interpretação durante a execução do código.

Com relação ao gerenciamento de memória pelo ambiente de execução, como descrito em [DRAKE et al. 2014], o sistema Android não oferece área de *swap* de memória, mas utiliza mecanismos de paginação e de mapeamento de arquivos.

Com o mecanismo de paginação, é utilizado o compartilhamento de páginas entre os processos. Isso permite que muitas páginas, alocadas para o código e os recursos do *framework* Android, sejam compartilhadas por todos os outros processos das aplicações.

Com mecanismo de mapeamento, a maioria dos dados de natureza estática de uma aplicação (*bytecode*, recursos e possíveis bibliotecas de código nativo) é mapeada no espaço de endereçamento de memória do processo da aplicação. Isso permite que haja um compartilhamento de dados entre os processos e também que páginas de memória possam ser descarregadas quando necessário. Esse compartilhamento de memória entre aplicações se

dá através da alocação pelo mecanismo de compartilhamento assíncrono denominado *Anonymous Shared Memory* (Ashmem). O *Ashmem* é uma modificação acrescida ao kernel Linux do Android para permitir que automaticamente se ajuste o tamanho dos caches de memória e se recupere áreas quando a memória total disponível estiver em baixa [YAGHMOUR 2013].

Dessa forma, qualquer área de memória de uma aplicação que é modificada (por alocação de novos objetos ou por alteração em mapeamentos) permanece residente na RAM e não pode sofrer *swap* de página. Para uma *heap* de objetos Java, a única maneira de se desalocar a memória utilizada por uma aplicação é liberar as referências de objetos que possam estar em uso, tornando a memória disponível para ação do GC. Já os arquivos que armazenam dados de natureza estática e que são mapeados para uma aplicação sem modificação podem sofrer *swap* para fora da RAM caso o sistema esteja precisando de memória disponível.

Em item posterior desse trabalho são detalhados aspectos relevantes para interpretação das estruturas de memória envolvidas no novo ambiente de execução (ART) e algumas diferenças com relação ao ambiente de execução anterior (DALVIK).

2.5. FRAMEWORK ANDROID

O *framework* Android é uma implementação do Java com bibliotecas adicionais, incialmente baseada no projeto de código aberto Apache Harmony[5], e que a partir da mais nova versão do Android, denominado Nougat, lançada em agosto de 2016, passou a ser baseada no projeto OpenJDK[6].

Na organização do *framework*, os pacotes de classe que fornecem a API sobre os serviços possuem nomenclatura prefixada com termo *android*, e as classes básicas, com *java*.

Em nível lógico, as classes JAVA de uma aplicação comum são compostas de quatro tipos de componentes: (1) *Activities*, que representam um tela com interface de usuário; (2) *Services*, que são executados em background sem interface de usuário, e são responsáveis

[5] Implementação de código aberto do Java desenvolvida pela Apache Software Foundation. Disponível em http://harmony.apache.org/

[6] Projeto de plataforma Java totalmente baseada em software livre e em código aberto. Disponível em http://openjdk.java.net/

por operações de longa duração ou executam tarefas para processo remotos; (3) *Content Providers*, que gerenciam dados compartilhados da aplicação em arquivos, banco de dados SQLite, repositórios na Internet ou qualquer outro meio de armazenamento disponível; (4) *Broadcast receivers*, que respondem a mensagens de broadcast do sistema como desligamento de tela, nível de bateria baixo, ou quando uma fotografia foi produzida. A ativação dos componentes dos tipos *Activities*, *Services* e *Broadcast Receivers* é realizada através de objetos de mensagens assíncronas denominados *Intents*.

2.6. INICIALIZAÇÃO DO SISTEMA ANDROID

Conforme Figura 2.3, o processo inicial de boot do *kernel* do Android não tem diferença com relação ao Linux padrão. Assim, após a inicialização do *kernel* e dos drivers contidos nele, o *kernel* inicia, no espaço de usuário, apenas o processo próprio do Android denominado *init*. Este processo é responsável por lançar todos os outros processos e serviços do sistema e conduzir operações críticas como uma reinicialização do sistema. Dentre os processos iniciados por ele, existe um que é chave para o funcionamento do Android denominado Zygote.

O processo Zygote é um tipo especial de *daemon* cujo trabalho é iniciar aplicações. O objetivo é a centralização dos componentes compartilhados por todas aplicações e a diminuição do tempo de inicialização. O processo *init* não inicia o Zygote diretamente. Ao invés disso, utiliza o comando `app_process` para que o processo Zygote seja inicializado pelo ambiente de execução, que, por usa vez, inicia a primeira máquina virtual do sistema e invoca o método `main()` do Zygote.

O processo Zygote só fica ativo quando uma nova aplicação vai ser iniciada. Para acelerar o processo de iniciação de uma aplicação, é feito carga na memória RAM das classes do *framework* Android e dos recursos associados. Nesse estado, o Zygote passar a escutar, via *socket*, requisições para o processo de abertura de novas aplicações.

Essa estratégia acelera a iniciação das aplicações, pois o *kernel* do Linux implementa a política de *copy-on-write* durante o *fork* de um processo, sendo produzida uma cópia exata do processo pai [MCHOES e FLYNN 2011]. O processo filho possui um mapeamento das páginas de memória do processo pai, sendo que os dados só são copiados para área de memória do processo filho caso haja escrita em alguma dessas páginas. No Android, como as classes e os recursos associados do *framework* não sofrem modificação durante a

operação normal do aparelho, as páginas do Zygote não serão copiadas para os processos filhos e isso independe do número de aplicações em execução no sistema, logo, em todo sistema apenas uma área de memória conterá as classes de sistema e os recursos associados.

Fonte: Adaptação de figura em [YAGHMOUR 2013]

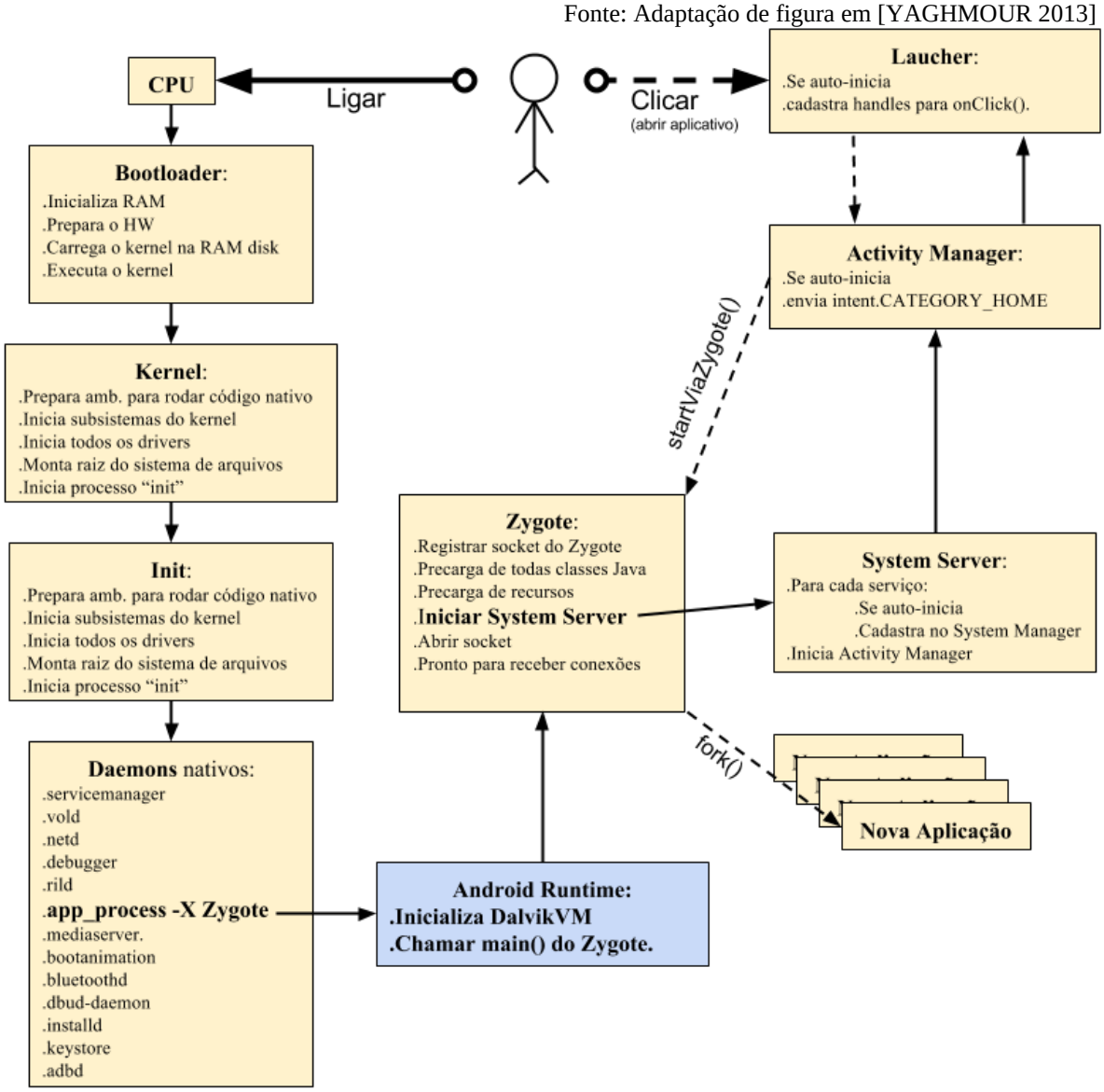


Figura 2.3 - Sequência de boot do Android

Conforme a Figura 2.3, nas ações de inicialização do Zygote antes da abertura do *socket*, é executado o *System Server* explicitamente para que ele inicie e registre todos os serviços, inclusive o denominado *Active Manager*, que ao receber o objeto de mensagem `Intent.CATEGORY_HOME`, executa a aplicação do sistema gráfico denominado *Launcher*, que disponibiliza a interface gráfica com os programas para o usuário. A partir daí, em algum momento quando o usuário clica para iniciar uma aplicação, o *Launcher* solicita ao *Activity*

Manager que inicie a aplicação, esse por sua vez encaminha para o processo Zygote, que faz um *fork* de si mesmo e inicia a nova aplicação para o usuário com as classes de sistema e os recursos associados prontos para uso e disponibilizados no espaço de memória virtual do aplicativo.

Assim, segundo [YAGHMOUR 2013], o ambiente de execução *Android Run-time* é um dos componentes essenciais do sistema operacional, sendo responsável, dentre outras operações, por iniciar o processo Zygote com o ambiente de máquina virtual, e gerenciar a execução das demais aplicações.

```
...
[dalvik.vm.dex2oat-Xms]: [64m]
[dalvik.vm.dex2oat-Xmx]: [512m]
[dalvik.vm.dexopt-flags]: [m=y]
[dalvik.vm.heapgrowthlimit]: [128m]
[dalvik.vm.heapmaxfree]: [8m]
[dalvik.vm.heapminfree]: [2m]
[dalvik.vm.heapsize]: [512m]
[dalvik.vm.heapstartsize]: [8m]
[dalvik.vm.heaptargetutilization]: [0.75]
[dalvik.vm.image-dex2oat-Xms]: [64m]
[dalvik.vm.image-dex2oat-Xmx]: [64m]
[dalvik.vm.isa.arm.features]: [div]
[dalvik.vm.stack-trace-file]: [/data/anr/traces.txt]
…
[persist.sys.dalvik.vm.lib.2]: [libart.so]
...
```

Figura 2.4 - Algumas propriedades do sistema relacionadas à DVM e ao ambiente de execução

Dentre os serviços ativados na inicialização, o *Property Service* é o responsável por manter (para cada inicialização) um mapeamento de memória com pares do tipo chave-valor referentes a configuração do sistema, congregando esses dados de arquivos de propriedades e outras fontes do sistema operacional. Muitos componentes do sistema operacional e do *framework* Android, inclusive o ambiente de execução ART, utilizam desses valores que incluem itens referentes a configuração da interface de rede, opções de rádio celular, e também configurações de segurança [DRAKE et al 2014]. Dentre esses itens, existem propriedades diretamente ligadas ao ambiente de execução e configurações da máquina virtual, por exemplo as listadas na Figura 2.4. Normalmente, essas propriedades podem ser

obtidas pelo comando *getprop* via *shell* Linux Android, ou programaticamente via código nativo através da biblioteca `libcutils`, ou pela classe de sistema `android.os.SystemProperties`. Como exemplo, o valor da propriedade definida pela chave com prefixo `persist.sys.dalvik.vm.lib`, define a biblioteca a ser utilizada pelo sistema como ambiente de execução, que até a versão 4.4 do Android era configurado como `libdvm.so`, e para o ambiente ART, das versões 5.0 e superiores, passou a ser `libart.so`, cujo código fonte foi base de estudo desse trabalho.

2.7. AMBIENTE DE EXECUÇÃO ART

O ambiente de execução ART opera sobre os mesmos aplicativos construídos para o *bytecode* do ambiente anterior, se diferenciando, essencialmente, na forma e no momento como o *bytecode* contido nos arquivos DEX é carregado, compilado e executado.

Como ilustrado na Figura 2.5, para esse novo ambiente de execução, a constituição de uma aplicação Android permanece como anteriormente, mudando o processo de instalação e carga a partir de um APK. Na porção inferior direita da figura é ilustrado o processo de carregamento para o ambiente ART, enquanto que na esquerda é ilustrado como era nos ambientes de execução anteriores, denominados com o mesmo nome da máquina virtual (DALVIK).

Observa-se na figura que o novo ambiente de execução é compatível com o formato de *bytecode* anterior e que, durante a instalação o código DEX, no ambiente ART, este é compilado pelo comando `dex2oat`, localizado em `/system/bin/dex2oat`, ao invés do comando `dexopt` de otimização de código, utilizado no ambiente de execução anterior. Por questão de desempenho, nas versões do Android anteriores à 4.4, os arquivos DEX sofrem otimização antes de serem interpretados pela máquina virtual. Esses arquivos são otimizados quando iniciados pela primeira vez. O produto dessa otimização é um arquivo denominado *Optimized DEX* (ODEX), que não é portável para os diferentes dispositivos e versões do Android.

Examinando o código fonte e o trabalho em [BACKES et al. 2016], percebe-se que a compilação de DEX para o código nativo pode operar dois tipos de compilação: *Quick*,

como compilador padrão, e *Portable*, utilizando o LLVM[7]. O tipo de compilador utilizado é definido pelo parâmetro `--compiler-backend` do comando `dex2oat`. Como os códigos DEX sofrem compilação durante a instalação das aplicações, caso haja uma restauração para configuração de fábrica ou uma atualização do sistema, há acréscimo no tempo de inicialização devido à necessidade de recompilação das aplicações.

Fonte: Adaptação de [GOOGLE 2016]

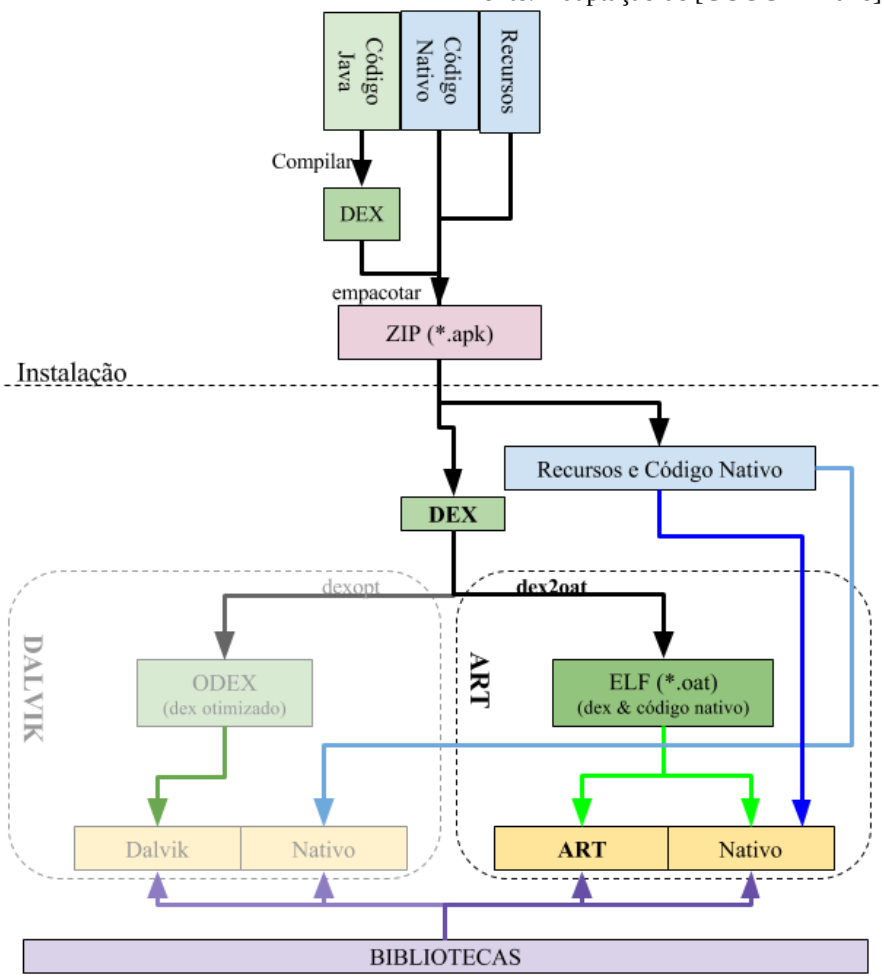


Figura 2.5 Carregamento de uma aplicação Android

Por isso, no código fonte do ART existem várias opções de compilação para pré-otimização de bibliotecas e aplicações, que podem ser definidas pelo fabricante de acordo com as estratégias adequadas para as capacidades do hardware, inclusive com opções de

[7]Conjunto de ferramentas de compilação modulares, capaz de trabalhar com várias linguagens, desenvolvido para otimizar processos de compilação, ligação e execução, que geralmente visam alta performance com baixo consumo de energia. Site do projeto:`llvm.org`

trabalhar apenas com a imagem do *framework* compilada previamente ou apenas com classes específicas. Também pode ser definido o modo de compilação levando em conta a priorização entre desempenho de execução ou espaço de armazenamento, ou mesmo evitar qualquer compilação e operar apenas com código interpretado durante a execução.

Na compilação *Quick*, utilizada por padrão pelo ambiente de execução ART na versão 5.0, o *bytecode* (linguagem de representação intermediária) é traduzido para uma outra linguagem intermediária de baixo nível e depois para código nativo, com a execução de algumas otimizações de código nesse processo. Na compilação *Portable*, é utilizado o LLVM na construção da linguagem intermediária, possibilitando que sejam feitas, com alta desempenho, várias otimizações na geração do código nativo, específicas para o dispositivo alvo.

O produto da compilação do *bytecode* são arquivos no formato *Executable and Linking Format*[8] (ELF) denominados OAT, que são armazenados no diretório `/data/dalvik-cache/<arch>`, onde `<arch>` é o nome da arquitetura alvo da compilação (para arquitetura ARM, `/data/dalvik-cache/arm`).

Com relação ao carregamento do código, o ART não utiliza uma área de cache para código como utilizado anteriormente pelo ambiente DALVIK interpretado. Esses arquivos OAT são mapeados para memória (via ashmem), permitindo paginação. Outro ponto é que, semelhante ao carregamento prévio de classes realizado pelo processo Zygote, o ART faz uma inicialização de um conjunto de classes durante o processo de compilação, onde se gera um arquivo que contém uma imagem de todas as classes e objetos que fazem parte do *framework* Android. Esse arquivo é mapeado na memória (na versão do Android 5.0 estudada esse arquivo ocupa aproximadamente 11 MiB) através do processo de inicialização do Zygote e, por ser paginável, permite operações de *swap* de página, o que acarreta que em um processo de aquisição de memória volátil possa não conter dados.

Esse arquivo de imagem do ART é denominado `boot.art`. O código compilado contido nos arquivos executáveis OAT faz referência direta a esse arquivo de imagem para chamada de métodos ou acesso a objetos do *framework* Android. Assim, ele contém basicamente objetos com ponteiros de endereçamento absoluto para objetos dentro da

[8]Especificação ELF contida no Linux Tool Interface Standard (TIS). Disponível em `https://refspecs.linuxfoundation.org/elf/elf.pdf`

própria imagem e para código nos arquivos OAT, sendo que nos próprios arquivos OAT, também existem ponteiros absolutos para métodos na imagem.

Fonte: Adaptação de [BECHTSOUDIS 2015]

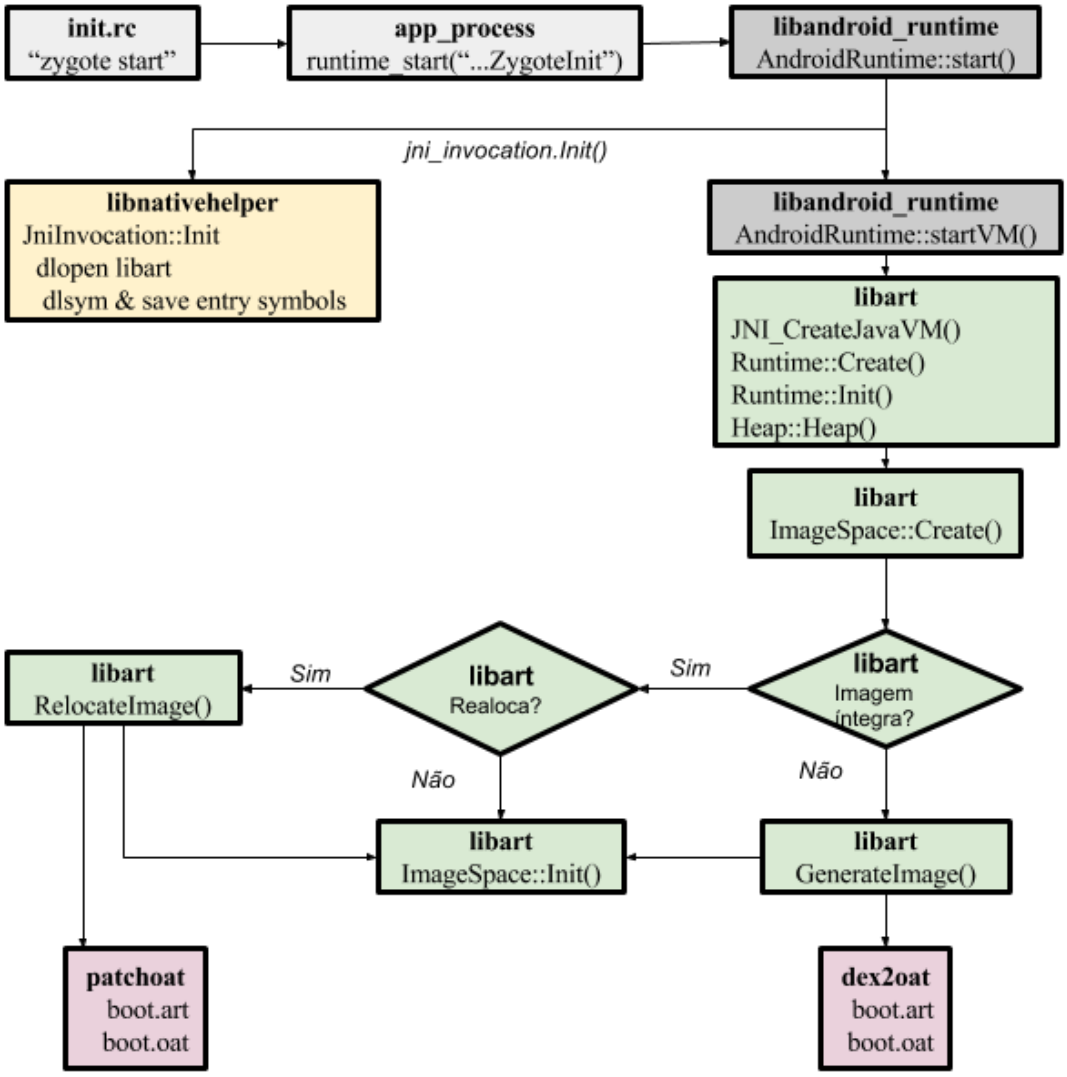


Figura 2.6 Processo de inicialização do ART

Como se observa, o arquivo de imagem é um ponto crítico de funcionamento e segurança do Android, dessa forma, no processo de inicialização do Android descrito anteriormente, existe um sub-processo para manutenção desse arquivo de imagem, ilustrado na Figura 2.6. Descrevendo sumariamente, no processo de inicialização *init* faz-se uma chamada para o ambiente de execução (*libandroid_runtime*) que por sua vez solicita carregamento da biblioteca do ART (*libart*) e que, baseado em parâmetro de configuração mantidos pelo serviço *System Properties*, é responsável por iniciar o ambiente de execução com a criação de um ambiente de execução de máquina virtual DVM, com a preparação da *heap* de objetos e procedendo com a manutenção do arquivo de imagem com relação a sua

integridade e com ajuste das referências de acordo com o parâmetro de realocação, que atua como mecanismo de segurança do tipo *Address Space Layout Randomization*[9] (ASLR).

2.7.1. ARQUIVOS DE IMAGEM ART E ARQUIVOS OAT

O arquivo de imagem, construído e mantido pelo ART, tem um formato próprio, cujo cabeçalho é detalhado na Tabela 3.2, cuja versão do tipo de imagem é definida no código fonte estudado como "009".

Os arquivos OAT são arquivos ELF contendo seções com dados específicos relacionados ao ambiente de execução Android. Conforme Figura 2.7, nesses dados existem cabeçalhos descrevendo a estrutura do arquivo OAT, bem como, *bytecodes* em estruturas de arquivos tipo DEX e códigos nativos. Dentro do arquivo OAT, são utilizadas três tabelas dinâmicas de símbolos denominadas `oatdata`, `oatexec` e `oatlastword`. Em `oatdata` estão contidos o cabeçalho OAT, descrito na Tabela 3.3, e os arquivos DEX, em `oatexec` estão contidos os códigos nativos compilados para cada método, e em `oatlastword`, estão contidos os 4 últimos bytes de código nativo gerado, funcionando como um marcador de final de seção.

Tabela 2.1 - Cabeçalho de um arquivo de imagem ART

Campo	Tipo	Descrição
Magic	ubyte[4]	Valor "art\n"
Version	ubyte[4]	Versão do tipo de imagem
image_begin	uint32	Endereço base da imagem
image_size	uint32	Tamanho da imagem (bytes)
image_bitmap_offset	uint32	Offset para o bitmap
image_bitmap_size	uint32	Tamanho do bitmap

[9] Técnica de segurança que compreende na randomização no posicionamento de áreas de endereçamento de um processo, geralmente no endereçamento do código base, das bibliotecas, da *heap* e da pilha. Presente no Android a partir da versão 4.0 conforme descrito em `https://source.android.com/security/enhancements/enhancements41.html`

oat_checksum	uint32	Checksum do arquivo boot.oat vinculado
oat_file_begin	uint32	Endereço do arquivo boot.oat vinculado
oat_data_begin	uint32	Endereço inicial do segmento “oatdata” do arquivo boot.oat vinculado
oat_data_end	uint32	Endereço final do segmento “oatdata” do arquivo boot.oat vinculado
oat_file_end	uint32	Endereço final do arquivo boot.oat vinculado
patch_delta	int32	Delta para reposicionamento do endereçamento do código
image_roots	uint32	Endereço para um array de objetos
compiler_pic	uint32	Indica se a imagem foi compilada com PIC (position-independent-code) ativado.

Fonte: Código-fonte em [AOSP 2014]

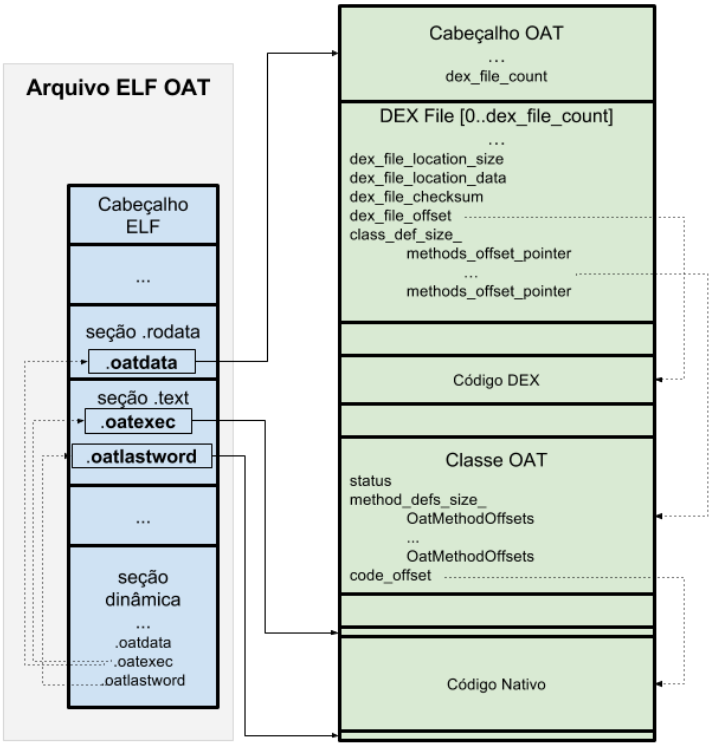


Figura 2.7 Estrutura do arquivo OAT

No cabeçalho OAT, está descrita a estrutura geral dos dados relacionados com a compilação e execução no ambiente ART. No campo *magic* existe o identificador “oat\n”,

seguido pela versão corrente do formato do arquivo, que para esse trabalho é definida no código fonte estudado como versão "039". O campo `instruction_set` indica qual arquitetura do conjunto de instruções utilizada como alvo de compilação (as arquiteturas suportadas então descritas na Tabela 3.4). O campo `dex_file_count` descreve a quantidade de arquivos DEX que foram compilados do APK original, o campo `executable_offset` aponta para seção de código nativo gerado (o mesmo ponteiro existente na tabela de símbolos dinâmica do ELF). O campo `image_path_delta` armazena o deslocamento definido para imagem ART com relação ao campo `image_begin`. Esse valor muda aleatoriamente (vide o componente *patchoat* na Figura 2.6) para cada processo de inicialização. Para o Android versão 5.0, o endereço de base para o deslocamento foi definido em 0x70000000. Essa informação pode ser explorada por *malwares* como vulnerabilidade do mecanismo de ASLR, podendo ser utilizada para alterações na seção `oatexec` do ELF (onde está armazenado o código nativo), deixando os dispositivos vulneráveis a ataques do tipo *Return Oriented Programming*[10] - ROP, conforme descrito em [SABANAL 2015b]. O campo `key_value_store` guarda metadados do arquivo OAT, como os parâmetros utilizados pelo comando `dex2aot` durante a criação do arquivo. Os demais campos com os sufixos `*_trampoline_offset` e `*_bridge_offset` são utilizados durante o processo de execução e foi constatado experimentalmente que geralmente possuem o valor zero no ambiente de execução emulado.

Tabela 2.2 - Cabeçalho de um arquivo OAT

Campo	Tipo	Descrição
Magic	ubyte[4]	Valor "oat\n"
Version	ubyte[4]	Versão do OAT
adler32_checksum	uint32	Checksum Adler-32 dos dados do cabeçalho
instruction_set	uint32	Conjunto de instruções da arquitetura
instruction_set_features	uint32	Máscara de bits para os recursos suportados pela arquitetura
dex_file_count	uint32	Quantidade de arquivos DEX contidos no OAT

[10] Conforme [DRAKE et al. 2014], técnica de ataque para execução de código arbitrário aproveitando código executável sem restrições já existente na memória.

executable_offset	uint32	Offset da seção de código executável desde o começo da oatdata
interpreter_to_interpreter_bridge_offset	uint32	Offset a partir do começo da oatdata até o stub interpreter_to_interpreter_bridge
interpreter_to_compiled_code_bridge_offset	uint32	Offset a partir do começo da oatdata até o stub interpreter_to_compiled_code_bridge
jni_dlsym_lookup_offset_	uint32	Offset a partir do começo da oatdata até o stub jni_dlsym_lookup
portable_imt_conflict_trampoline_offset	uint32	Offset a partir do começo da oatdata até o stub portable_imt_conflict_trampoline
portable_resolution_trampoline_offset	uint32	Offset a partir do começo da oatdata até o stub portable_resolution_trampoline
portable_to_interpreter_bridge_offset	uint32	Offset a partir do começo da oatdata até o stub portable_to_interpreter_bridge
quick_generic_jni_trampoline_offset	uint32	Offset a partir do começo da oatdata até o stub quick_generic_jni_trampoline
quick_imt_conflict_trampoline_offset	uint32	Offset a partir do começo da oatdata até o stub quick_imt_conflict_trampoline
quick_resolution_trampoline_offset	uint32	Offset a partir do começo da oatdata até o stub quick_resolution_trampoline
image_patch_delta	int32	Delta para o endereço de realocação
image_file_location_oat_checksum	uint32	Checksum Adler-32 dos dados do cabeçalho do arquivo boot.oat
image_file_location_oat_data_begin	uint32	Endereço virtual da seção oatdata do arquivo boot.oat
key_value_store_size	uint32	Tamanho do campo key_value_store
key_value_store	ubyte[key_value_store_size]	Dicionário contendo informações como a linha de comando utilizada para geração do arquivo oat, arquitetura do dispositivo de compilação, etc.

Fonte: Código-fonte em [AOSP 2014]

Tabela 2.3 - Código das arquiteturas suportadas

Conjunto de Instruções	Valor	Descrição
kNone	0	Não especificado
kArm	1	ARM
kArm64	2	ARM 64-bits
kThumb2	3	Thumb-2
kX86	4	X86
X86_64	5	X64
kMips	6	MIPS
kMips64	7	MIPS 64-bits

Fonte: Código-fonte em [AOSP 2014]

2.7.2. GERENCIAMENTO DE MEMÓRIA

Para sistemas anteriores ao ART, o ambiente de execução dividia o espaço de memória para uso da DVM em três áreas principais: *heap* e pilha de objetos Java, bitmaps e dados de código nativo. A *heap* de objetos Java era utilizada para manutenção dos objetos em uso pela aplicação. O tamanho desse espaço era definido no momento da iniciação da máquina virtual onde era especificada a quantidade mínima e máxima de espaço utilizado por meio de parâmetros definidos nas propriedades do sistema, geralmente configurado de fábrica de acordo com as capacidades do hardware do aparelho. A área de bitmaps era utilizada para armazenamento dos bitmaps da aplicação. O espaço destinado para dados de código nativo era utilizado para alocação de memória pelo código nativo da aplicação.

No novo ambiente de execução ART, a gerência de memória divide o espaço de memória virtual utilizado pela DVM em espaço de alocação de objetos Java da aplicação (*heap*), espaço para imagem de objetos do *framework* Android, contendo classes do processo Zygote e objetos Java previamente alocados, e espaço de alocação de objetos grandes. Essas áreas são mapeadas em memória conforme modelo simplificado ilustrado na Figura 2.8, onde os três primeiros são organizados para espaço contínuo de endereçamento, enquanto que para o último espaço é definida uma coleção de endereços discretos descontínuos.

Entre o espaço de imagem de objetos do *framework* e o espaço do Zygote são armazenadas as classes do *framework* Android. Nessa área é feito um mapeamento para o arquivo `system@framework@boot.art@classses.oat`. Este arquivo é composto de arquivos DEX do *framework* que são compilados durante a iniciação do sistema e onde é feita uma tradução do caminho dos componentes (*classpath*) para o espaço de imagem com objetos pré-carregados. A partir do arquivo `system@framework@boot.art@classes.dex`, que contém todos os objetos que serão pré-carregados, o arquivo `system@framework@boot.art@classses.oat` é montado, e desde que não haja nenhuma mudança nas classes de *framework* que necessite a recompilação, a cada iniciação, o sistema simplesmente mapeia o arquivo com os objetos diretamente na memória. Isso exige que a primeira iniciação do sistema demande um certo tempo devido à preparação do arquivo com os objetos do *framework.*

Como os objetos contidos no *framework* podem trocar mensagens entre si, o endereçamento é fixado no momento de carregamento do *framework*, como foi detalhado na Figura 2.8. O endereço de carregamento é armazenado no início do arquivo `system@framework@boot.art@classses.oat`, no cabeçalho da imagem ART. O mapeamento para o arquivo `system@framework@boot.art@classes.dex` é estabelecido no endereço de memória contíguo ao fim da imagem.

Como abordado anteriormente sobre o processo de inicialização, devido ao uso do *fork* na criação de novas aplicações, o espaço de alocação do processo Zygote é compartilhado entre as aplicações, enquanto que a área de alocação comum é destinada apenas para o processo no qual ela pertence.

Além desses espaços descritos, são utilizadas estruturas diretamente relacionadas com a manutenção de referências entre as áreas de memória para a ação do sistema de GC como tabela *ModUnion*, tabela *CardTable*, bitmaps de alocação, mapeamento de objetos grandes, e pilhas de objetos.

Também é observado no código fonte AOSP, que as estruturas relacionadas à manutenção do sistema GC dependem do plano de GC que está ativo. O plano de GC geralmente é definido pelo fabricante de acordo com as características intrínsecas do dispositivo. É possível modificar o plano utilizado, através de uma compilação específica ou através de um usuário privilegiado, alterando-se a propriedade denominada `dalvik.vm.gctype` ou através do uso do parâmetro `-Xgc: option` na criação da DVM. Assim, de acordo com o tipo de GC configurado, são utilizados alocadores de memória

específicos. O plano do tipo *Concurrent Mark Sweep* (CMS) é definido por padrão, e nesse modo de operação o alocador utilizado é *Runs-of-Slots-Allocator* (RosAlloc) para objetos mutáveis (que estão localizados na *heap* de objetos da aplicação) e DlMalloc[11] para objetos imutáveis (p.ex. localizados na área de endereçamento contínuo).

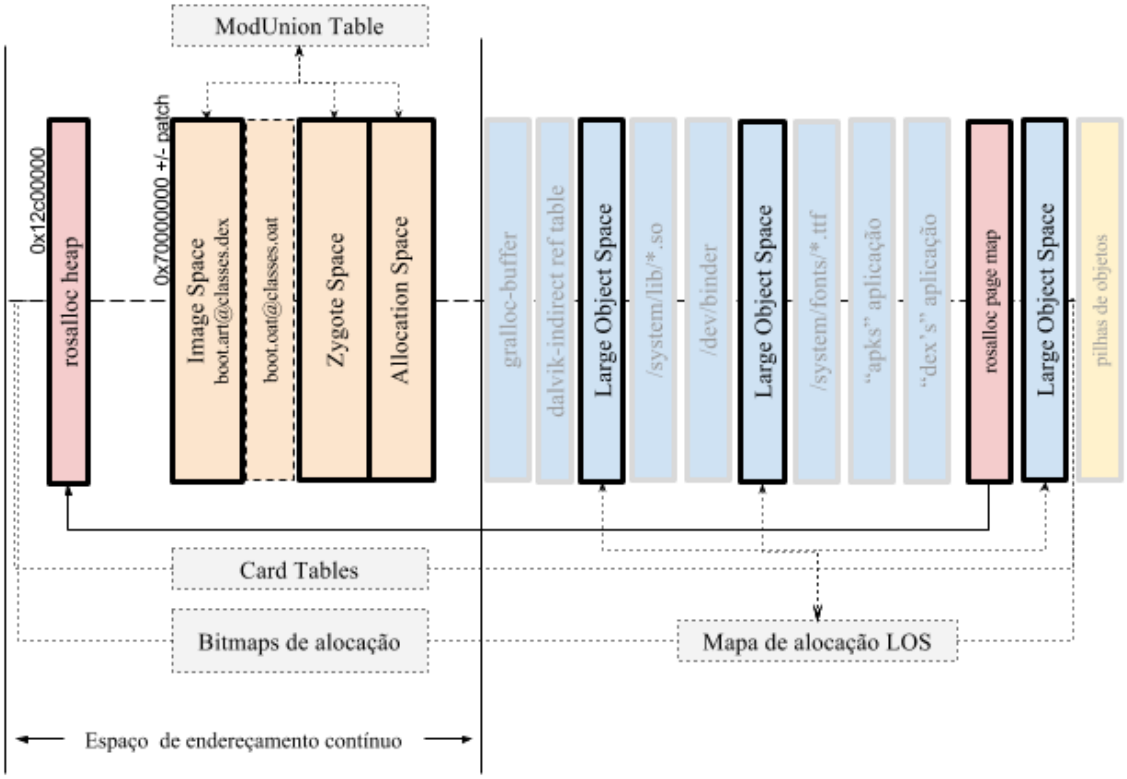


Figura 2.8 Modelo simplificado de algumas estruturas de memória mapeadas para uma aplicação

2.7.3. ALOCADOR DE MEMÓRIA ROSALLOC

O alocador RosAlloc é o alocador de memória que surgiu com o ambiente de execução ART. Ele é o alocador principal responsável pelo espaço de memória *heap* destinado aos objetos Java da aplicação. Ele possui *slots* classificados de acordo com o tamanho dos objetos, com objetivo de reduzir a fragmentação, permitir um bloqueio com granularidade no nível do *slot* e possibilitar o uso de *buffers* de alocação por meio de *Thread Local Allocation* (TLA). O espaço de objetos grandes, definidos em código como dados maiores ou iguais a 12KiB são separados em um espaço de endereçamento virtual descontínuo denominado *Large Object Space* (LOS). Com isso, é dada a liberdade para o

[11] Alocador de memória de uso geral pelas bibliotecas padrão Linux.

kernel de encontrar um endereço conveniente para mapear cada espaço destinado a objetos grandes, dando maior eficiência nas alocações.

A técnica de alocação do RosAlloc é semelhante a técnica SLAB utilizada para alocações de memória pelo *kernel* Linux (a partir da versão 2.2), no qual a ideia básica é a de que a memória é dividida em vários compartimentos para blocos de memória do mesmo tamanho e, quando é solicitada uma alocação, é disponibilizado um bloco livre que faça parte do compartimento destinado a blocos do mesmo tamanho do solicitado [SILBERSCHATZ et al. 2008]. Assim, o alocador RosAlloc utiliza um mapa das páginas de alocação (definida no código fonte com tamanhos de 4KiB) contendo a marcação do tipo de página e agrupamentos que obedecem à codificação descrita na Tabela 3.5. Conforme mencionado, cada página contém um conjunto de *slots* e em cada um é feita alocação dos dados de um objeto de acordo com seu tamanho. Dependendo do tamanho do objeto é definido em qual conjunto de páginas ele será alocado.

Tabela 2.4 - Codificação do tipo de página de alocação para uso do RosAlloc

Tipo	**Código**	**Descrição**
kPageMapReleased	0	Liberada para o S.O.
kPageMapEmpty	1	Vazia, mas provavelmente com fragmentos de dados (dirty)
kPageMapRun	2	Início de uma série de páginas (Runs) de alocação normal.
kPageMapRunPart	3	Parte não inicial de uma série de páginas de alocação normal.
kPageMapLargeObject	4	Início de uma série de páginas para alocação de objetos grandes.
kPageMapLargeObjectPart	5	Parte não inicial de uma série de páginas para locação de objetos grandes.

Fonte: Código-fonte em [AOSP 2014]

Os metadados contendo o tipo de página de alocação ficam registrados em um arquivo mapeado para essa *heap* principal do processo em `/dev/ashmem/dalvik-rosalloc page map`. Existe a possibilidade de que, quando operando com plano de GC com compactação ativa, seja criada *heap* de backup, localizada contígua a *heap* principal, e um mapeamento de página de alocação associada, localizado em endereço posterior ao mapeamento principal.

Os dados de alocação da *heap* de objetos ficam definidos próximos do endereço virtual mais baixo do processo, iniciando a partir dos 300 MiB, conforme código fonte (em `art\rutnime\gc\heap.cc`). Conforme descrito no código fonte (em `runtime/gc/allocator/rosalloc.h`), no cabeçalho de cada página, é definida a classe do agrupamento de páginas (*brackets*) que estabelecem o número de páginas do agrupamento. Na Tabela 3.6 se tem a relação entre o índice de classificação e a quantidade de páginas.

A quantidade de *slots* por página é definida em função do tamanho do agrupamento de páginas e do alinhamento de bytes utilizado (que depende da arquitetura do dispositivo alvo). A quantidade de *slots* por agrupamento é determinada em função do tamanho do agrupamento e pelo tamanho final do cabeçalho do agrupamento, ajustado em função do tamanho do *bitmap* de alocação de *slots* e do alinhamento de bits para arquitetura alvo. Essas informações estão presentes no cabeçalho do agrupamento conforme Tabela 3.7.

Tabela 2.5 - Relação entre o índice de classificação e a quantidade de páginas de um agrupamento de páginas.

Classe do agrupamento (bracket_idx)	Quantidade de páginas
< 4	1
< 8	2
< 16	4
< 32	8
32	16
> 32	32

Fonte: Código-fonte em [AOSP 2014]

Tabela 2.6- Composição de uma página inicial (kPageMapRun) de um grupo de páginas de alocação

Campo	Tamanho	Descrição
magic_num_	unsigned char	Utilizado em processo de debug
size_bracket_idx_	unsigned char	Índice classificador do agrupamento
is_thread_local_	unsigned char	Verdadeiro se esse segmento (run) é utilizado

		como um segmento thread-local
to_be_bulk_freed_	unsigned char	Utilizado como flag em processo de gerenciamento de memória (bulk free)
first_search_vec_idx_	unsigned int	Índice do primeiro vetor de bitmap que contém um slot disponível
alloc_bit_map_	Calculado = n slots + bytes de alinhamento	Bitmap de alocação para cada slot em uso
slot[0]	tamanhoSlot (tamanho_pagina, size_bracket_idx_, header)	
...	...	
slot[n]	...	

Fonte: Código-fonte em [AOSP 2014]

No interior de cada *slot* alocado, como se espera o armazenamento de dados referentes a objetos Java, os dados devem iniciar com um endereço de 4 bytes (para uma arquitetura de 32-bits) referentes a classe na qual pertence o objeto, sendo possível, aplicando as informações de definição da classe, interpretar os dados do objeto.

As informações de definição de uma classe, se forem objetos de classes raízes, podem ser recuperadas diretamente por meio do endereço mapeado do arquivo imagem ART mapeado no vetor armazenado no campo denominado `image_roots` (vide Tabela 3.2). Esses objetos de classes raízes estão descritas na Tabela 3.8 conforme a nomenclatura do projeto Apache Harmony (base de implementação das classes Java do projeto Android) e na ordem mesma presente no vetor de armazenamento do ambiente de execução (código fonte AOSP, art run-time: `art\runtime\class_linker.cc`).

Tabela 2.7 - Classes raízes da imagem do *framework* Android

Nome do descritor de classe	Descrição
`Ljava/lang/Class;`	Representação de uma classe
`Ljava/lang/Object;`	Representação de um objeto
`[Ljava/lang/Class;`	Array de classes
`[Ljava/lang/Object;`	Array de objetos

`Ljava/lang/String;`	Representação de uma String
`Ljava/lang/DexCache;`	Representação de um cache para resolução de strings, campos, métodos e classes de um arquivo DEX.
`Ljava/lang/ref/Reference;`	Representação de uma classe abstrata para descrever o comportamento de objetos do tipo referência.
`Ljava/lang/reflect/ArtField;`	Representa o campo de uma classe.
`Ljava/lang/reflect/ArtMethod;`	Representa o método de uma classe.
`Ljava/lang/reflect/Proxy;`	Representação de uma classe ou instância para criação dinâmica de classes do tipo proxy.
`[Ljava/lang/String;`	Array de String
`[Ljava/lang/reflect/ArtField;`	Array de ArtFiled
`[Ljava/lang/reflect/ArtMethod;`	Array de ArtMethod
`Ljava/lang/ClassLoader;`	Representa classe responsável pela carga de classes e recursos de um repositório.
`Ljava/lang/Throwable;`	Representa uma superclasse de todas as classes que podem ser lançadas pela máquina virtual no controle de exceções.
`Ljava/lang/ClassNotFoundException;`	Representa uma classe que é lançada quando um class loader não consegue encontrar uma classe.
`Ljava/lang/StackTraceElement;`	Representa uma classe referente item da pilha de exceção.
`Z`	Representação do tipo primitivo boolean
`B`	Representação do tipo primitivo byte
`C`	Representação do tipo primitivo char
`D`	Representação do tipo primitivo double
`F`	Representação do tipo primitivo float
`I`	Representação do tipo primitivo int
`J`	Representação do tipo primitivo long
`S`	Representação do tipo primitivo short
`V`	Representação do tipo primitivo void
`[Z`	Array da representação do tipo primitivo boolean
`[B`	Array da representação do tipo primitivo byte
`[C`	Array da representação do tipo primitivo char
`[D`	Array da representação do tipo primitivo double
`[F`	Array da representação do tipo primitivo float
`[I`	Array da representação do tipo primitivo int
`[J`	Array da representação do tipo primitivo long
`[S`	Array da representação do tipo primitivo short
`[Ljava/lang/StackTraceElement;`	Array de StacktraceElement

Fonte: Código-fonte em [AOSP 2014]

2.8. SEGURANÇA

Conforme analisado por [WÄCHTER e GRUHN 2015] a segurança dos dispositivos móveis, cada vez mais rígida, influencia na eficácia das técnicas de aquisição de dados da memória volátil. Como descrito no site oficial do projeto Android, depois de instalado em um dispositivo, cada aplicativo no Android é ativado em sua própria área de segurança com as seguintes características:

- Sendo um sistema multiusuário Linux, cada aplicativo tem um usuário diferente.
- Por padrão, o sistema atribui a cada aplicativo um ID de usuário do Linux exclusivo (o ID é usado somente pelo sistema e é desconhecido para o aplicativo). O sistema define permissões para todos os arquivos em um aplicativo, de modo que somente o ID de usuário atribuído àquele aplicativo pode acessá-los.
- Cada processo tem seu próprio ambiente de máquina virtual, portanto o código de um aplicativo é executado isoladamente de outros aplicativos.
- Por padrão, cada aplicativo é executado em seu próprio processo Linux. O Android inicia o processo quando é preciso executar algum componente do aplicativo; em seguida, encerra-o quando não mais é necessário ou quando o sistema precisa recuperar memória para outros aplicativos.
- É impossível fazer com que dois aplicativos compartilhem o mesmo ID de usuário do Linux, caso em que eles são capazes de acessar os arquivos um do outro. Para preservar os recursos do sistema, os aplicativos com o mesmo ID de usuário também podem ser combinados para serem executados no mesmo processo Linux e compartilhar o mesmo ambiente de execução (também é preciso atribuir o mesmo certificado, localizado no arquivo APK e cuja chave associada é utilizada para assinar digitalmente o arquivo dos aplicativos).
- Um aplicativo pode solicitar permissão para acessar dados de dispositivo como contatos do usuário, mensagens SMS, memória secundária (geralmente um cartão SD), câmera, Bluetooth. Para o Android em versões inferiores a 6.0, todas as permissões de aplicativo devem ser concedidas pelo usuário no momento da instalação.

Com isso, para segurança dos dados dos aplicativos, o Android implementa o princípio do privilégio mínimo. Ou seja, cada aplicativo, por padrão, tem acesso somente

aos componentes necessários para a execução do seu trabalho. Isso cria um ambiente de segurança em que o aplicativo não pode acessar partes do sistema para o qual não tem permissão. Como é uma distribuição Linux, existe o isolamento de processos onde cada processo possui seu espaço de endereçamento isolado dos outros, assim um processo não pode acessar diretamente a memória de outro processo.

Para a versão 5.0, conforme site do projeto AOSP, foram acrescidos novos mecanismos de segurança como criptografia padrão do tipo *Full Disk Encrytion* (FDE), e a exigência que toda ligação de código dinâmico seja do tipo relativo, *Position-Independent Code* (PIC), reforçando os mecanismos de ASLR. Além desses mecanismos, conforme descrito por [ARON e HANÁCEK 2015], foi ampliado o uso do controle mandatório de acesso através do SELINUX[12] para todos domínios do sistema, definindo um policiamento mais aprofundado dos recursos.

Todos esses mecanismos de segurança dificultam que um investigador execute qualquer tipo ação, inclusive uma simples extração de dados de um arquivo cujo armazenamento é restrito a uma aplicação, assim como um processo mais complexo, como a aquisição de dados brutos da memória volátil. Como para a maioria dos exames forenses, é preciso uma forma de se ter acesso irrestrito aos dados, um caminho para se ter esse acesso é por meio do ganho de privilégios para o usuário que está operando o sistema.

O processo de ganho de privilégios no Android é denominado "rooteamento", uma derivação do termo *root*, que define o nome da conta de super usuário do Linux. Essa conta tem direitos e permissões sobre todos os arquivos e programas em um sistema, possibilitando controle total sobre o sistema operacional. Como mencionado, os sistemas Android por padrão não permitem a escalada de privilégios para esse tipo de usuário e mantém diversos sistemas de segurança para restringir ao máximo esse tipo de ação.

O sucesso no "rooteamento" de um dispositivo está diretamente ligado às características do sistema desenvolvido pelo fabricante de um determinado dispositivo e muitas vezes exigem a exploração concomitante de vulnerabilidades existentes em várias camadas do sistema. No processo de "rooteamento" o aparelho pode ser reinicializado levando à perda dos dados voláteis que podem ser de interesse investigativo, porém existem aparelhos que permitem o denominado "soft root", que possibilita a obtenção da escalada

[12]SELINUX é uma implementação de controles mandatórios de acesso para o Linux. `https://selinuxproject.org/page/Main_Page`

temporária (até o próximo boot) de privilégio do usuário sem necessidade de reinicialização [DRAKE et al. 2014].

Como opção para extração de dados visando contornar mecanismos de segurança como telas de bloqueio com senha e cifragem de partições de dados, a técnica descrita em [HILGERS et al. 2014] demostra que mesmo com a reinicialização do aparelho podem haver informações de interesse forense, como credenciais ou outras informações de natureza efêmera utilizadas pelos aplicativos instalados e mantidas apenas na memória RAM, que possam ser recuperadas.

Para validação da técnica proposta nesse trabalho, foi feita a aquisição de dados da memória RAM por meio da técnica desenvolvida em [SYLVE et. al. 2012] tendo como premissa a execução por um usuário com poderes irrestritos (tanto para ambiente emulado como para aparelho real “rooteado”).

3. TÉCNICA DE ANÁLISE DE OBJETOS

Conforme exposto, no ambiente de execução ART de uma aplicação, existem arquivos mapeados na RAM contendo: informações sobre propriedades do sistema, *framework* Android, *heap* de objetos Java, mapeamento dos objetos utilizados pelo alocador de memória, e definições de classes e executáveis compilados a partir dos arquivos DEX da aplicação contidos nos arquivos OAT.

Partindo de uma extração total de memória RAM, a técnica proposta neste trabalho, detalhada na Figura 3.1, é para recuperação e análise de dados de objetos Java na *heap* através da varredura do mapeamento mantido pelo alocador de memória, baseada na premissa de que a partir de uma extração total de memória volátil é possível recuperar páginas de dados referentes a esses arquivos, pois, de acordo com o tipo de página (orientado pelo arquivo de mapeamento mantido pelo alocador) e dos dados do cabeçalho de cada uma, é possível recuperar os *slots* e, com a descrição adequada da classe do objeto alvo e suas referências para hierarquia de classes da imagem do *framework* Android, decodificar os dados dos objetos Java.

Para iniciar, é feita a identificação do endereço da imagem do *framework* Android e de suas classes *root*, em seguida é construída uma lista contendo as referências dos objetos da *heap*, a partir da varredura e decodificação dos slots dos agrupamentos, de acordo com o tipo de página descrita no arquivo de mapeamento mantido pelo alocador RosAlloc. Essa lista contém dados de objetos relativos à localização do objeto (endereço, página, *bracket* e *slot*), à classe ancestral do objeto, identificadores da classe do objeto no DEX e aos demais dados brutos ou textuais decodificáveis diretamente (objetos do tipo *String* ou *char array*) contidos no objeto.

Com essa lista de referências construída, é possível procurar por objetos específicos a partir dos seus endereços, por referências a sua classe, por referência a seu identificador nos arquivos DEX e, caso seja um objeto do tipo texto, por seu próprio conteúdo diretamente decodificado. Dessa forma, fica possível localizar uma característica do objeto e ascender na hierarquia de classes ao qual pertence recuperando os dados do objeto e de outros objetos interrelacionados. A decodificação dos dados de objetos pode ser realizada a partir do percorrimento das referências através da hierarquia de classes (análogo ao processo de programação por reflexão), utilizando informações de leiaute de memória obtidas por

decompilação do código da aplicação ou pelas informações de classes do *framework* Android obtidas do código fonte do AOSP (*java.lang.Class*, *java.lang.String*, etc.).

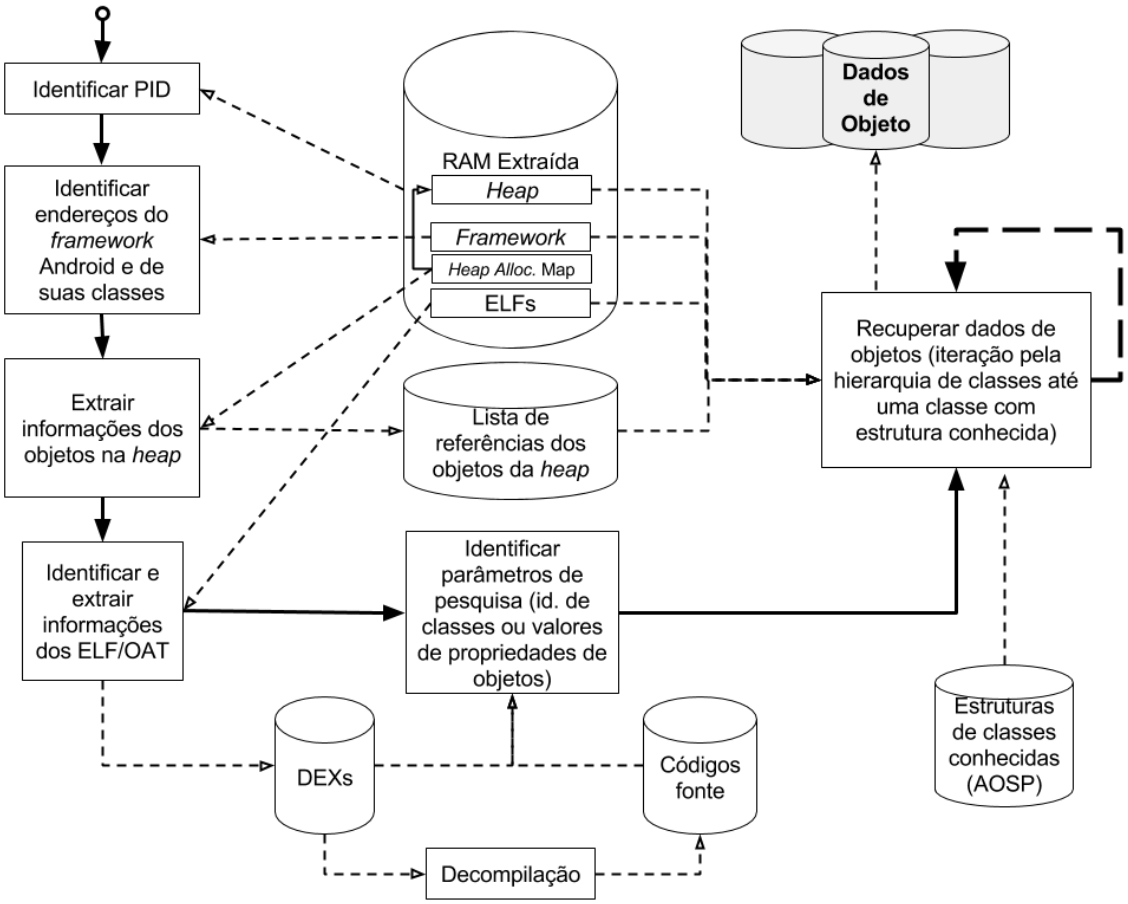


Figura 3.1- Técnica de análise de objetos da heap do alocador RosAlloc

Na Figura 3.2 é ilustrada uma sequência para um processo genérico de recuperação de uma *String* arbitrária como propriedade de um Objeto X. Do *slot* onde estão situadoa os dados do Object X, é possível percorrer a hierarquia de classes decodificando dados utilizando o leiaute das classes conhecidas cujos endereços na memória podem ser obtidos do cabeçalho da imagem do *framework* Android. Inicialmente (passo 1) é feita busca pela classe mãe da instância do objeto alvo (Object X) a partir do endereço existente nos primeiros bytes do *slot.* Em seguida (passo 2) é feita recuperação dos dados da classe ancestral (Class XParent) e seus dados são interpretados com base nas informações de leiaute presentes na imagem do *framework* Android (Class java.lang.Class). Dentres esses dados decodificados, é possível recuperar (passo 3) instância do objeto contendo *array* de objetos contendo os dados referentes às propriedades daquela classe (Object ArtFieldX[]). Para cada objeto que armazena dados das propriedades (Object ArtFieldX) é possível decodificar os dados (passos 4 a 7) baseado nas informações de leiaute recuperadas na

imagem do *framework* Android para classe mãe (Class ArtField) que contém o offset para os endereços contendo os dados das propriedades nos dados do objeto alvo (Object X). No endereço de uma propriedade do tipo *java.lang.String* hipotética, a decodificação é direta (passos 8 e 9) utilizando as informações de leiaute presentes na imagem do *framework* Android para a classe mãe (Class java.lang.String).

Devido à paginação provida pelo uso do mecanismo de compartilhamento *ashmem*, esse processo de busca da classe ancestral na extração de memória pode buscar endereços cuja página tenha sofrido *swap*, impossibilitando a decodificação completa dos dados do objeto.

Para operacionalização do processo de análise dos dados foi utilizado o conjunto de ferramentas disponíveis no *framework* Volatility. Esse *framework* (na versão 2.4), descrito em [LIGH et al. 2014], disponibiliza ferramentas e mapeamentos de estruturas de dados com suporte para plataforma Linux na arquitetura ARM 32-bits, possibilitando a recuperação de informações do ambiente Linux, como lista de processos e mapeamentos de memória, dentre outras.

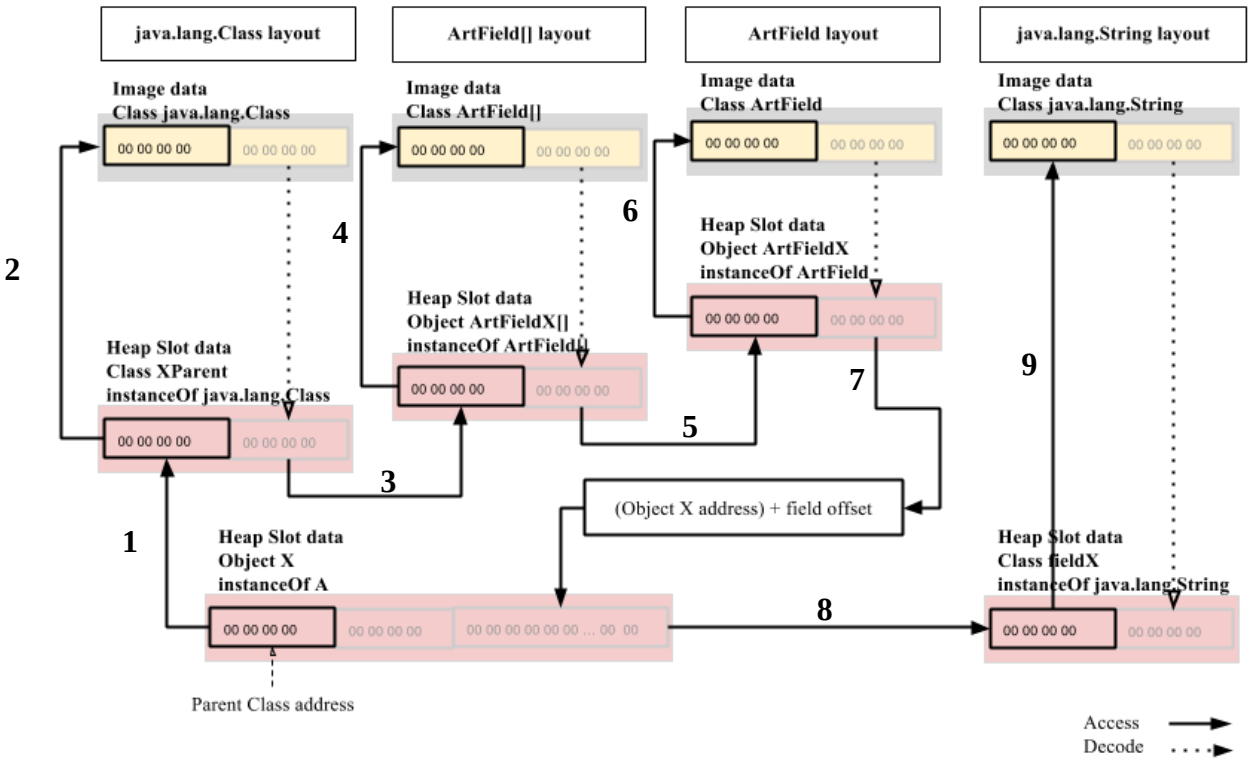


Figura 3.2 - Exemplo de processo de decodificação de dados de um Objeto X.

Para automatização da técnica, foi construído um conjunto de extensões para o *framework* Volatility que permitem a recuperação de informações além das obtidas do

kernel Linux, obtendo dados sobre o ambiente de execução e os objetos Java alocados. Para recuperação das estruturas de dados do ambiente de execução, foram criados mapeamentos para interpretação dos dados dos arquivos ART, OAT, DEX, classes do *framework* Java, estruturas de páginas da *heap* e propriedades do sistema. Para o processo de recuperação dos dados da extração de memória, foram construídas ferramentas para recuperação das propriedades do ambiente de execução, para localização dos arquivos OAT, para decodificação de dados dos arquivos DEX, para extração de objetos Java da *heap* e para decodificação dos dados dos objetos da *heap* e da imagem do *framework* Android.

A arquitetura do *framework* Volatility e das ferramentas construídas permite a atualização e a inclusão de novos mapeamentos, possibilitando adaptação para outras arquiteturas ou alterações em versões futuras do Android.

3.1. AQUISIÇÃO DE DADOS

Para proceder com a análise dos dados é preciso que previamente se tenha acesso aos dados da memória através de alguma técnica de aquisição. Até o momento, são conhecidas duas formas de aquisição: direta (no próprio aparelho) ou remota. Para dispositivos Android, as técnicas de análise remota são desconhecidas, logo o enfoque desse trabalho restringe-se às técnicas diretas.

Como mencionado, em [SYLVE et al. 2012] foi feito um estudo sobre vários métodos de aquisição direta onde é afirmado que a técnica que utiliza o dispositivo Linux `fmem` (que cria um dispositivo de caractere para ler os dados da memória) é considerada inadequada, pois faz uso de funções do *kernel* que não estão disponíveis para a maioria das plataformas ARM utilizadas no Android. Outra possibilidade de extração analisada se utiliza da implementação do comando Linux `dd`, mas esta não funciona de forma genérica, pois nesse caso, o sucesso depende muito da implementação do dispositivo. Outra técnica discutida como alternativa ao `dd` é a denominada *crash*, mas ambas utilizam uma solução de execução dividida entre o espaço de memória do *kernel* e do usuário, e como demonstrado essas técnicas são invasivas e preservam no máximo 80% da memória original.

A solução genérica de [SYLVE et al 2012] criada para qualquer sistema Linux é através do LiME utilizando um módulo de *kernel* carregável - *Loadable Kernel Module* (LKM) compilado especificamente para o *kernel* do dispositivo alvo. Observa-se que esta abordagem esbarra em mecanismos de segurança presente nos Linux atuais que

impossibilitam a construção de um módulo *kernel* genérico que possa ser carregado adequadamente no *kernel*. Para fins forenses, uma solução é a construção de um conjunto de módulos de *kernel* compilados paras as diversas distribuições Android que possuam os códigos fontes disponibilizadas pelos respectivos fabricantes.

Até o presente momento, desconhece-se técnica de extração de memória RAM baseada em JTAG[13] ou técnicas similares baseadas em acesso direto ao hardware.

Dentro das opções e limitações de técnicas de aquisição estudadas, a técnica em [SYLVE et al 2012] foi escolhida para este trabalho, por ser considerada viável para muitos casos e apresentar uma alta taxa de preservação dos dados originais. Como o *framework* Volatility e as ferramentas construídas para o processo de análise são agnósticas para o processo de aquisição, a técnica proposta pode ser utilizada com outras formas de aquisição de memória RAM.

No Apêndice A, foram descritos os procedimentos necessários para aquisição de dados da memória RAM utilizados na avaliação experimental desse trabalho.

3.2. FRAMEWORK VOLATILITY

Como já mencionado, o *framework* Volatility, disponibilizado sob licença de código aberto GNU e construído sobre a linguagem de programação Pyhton, é composto de um conjunto de ferramentas que possibilitam a extração de artefatos de dados de memória volátil. O projeto Volatility, na versão 2.4, disponibiliza ferramentas com suporte para plataforma Linux na arquitetura ARM[14].

Através de uma extração de memória volátil, as ferramentas disponibilizadas pelo *framework* permitem recuperar informações sobre tabela de processos, conexões de dados ativas, mapeamentos de memória de cada processo, e módulos de *kernel* ativos.

O *framework* disponibiliza uma arquitetura flexível e uma API que permitem a construção de extensões que possibilitam incrementar funcionalidades para suportar a recuperação de artefatos provenientes de extrações de memória e de outros sistemas operacionais.

---

[13] A JTAG é uma interface de *hardware* definida pela norma IEEE 1149.1, a qual é utilizada para o teste de circuitos eletrônicos.

[14] O *framework*, na sua versão 2.4, pode ser baixado através do site http://www.volatilityfoundation.org.

Com suporte recente desse *framework* para plataforma ARM, existem ferramentas para Linux que possibilitam a recuperação de diversas informações da camada do *kernel* de uma extração Android, conforme ilustrado na Tabela B.1.

Como requisito, é necessário a instalação do ambiente Python 2.6 ou mais atual, mas não as versões 3.x[15].

3.2.1. CONSTRUÇÃO DO PERFIL DO KERNEL ALVO

Antes de utilizar o *framework* para análise de uma extração de memória, é necessária a criação de um perfil (*profile*) para o sistema operacional alvo. O perfil contém basicamente informações das estruturas de dados (*vtypes*), ajustes para decodificação (*overlays*) e classes de objetos para uma determinada versão de sistema operacional e arquitetura de hardware. No caso de sistemas operacionais tradicionais como as diferentes versões do sistema operacional Windows, o Volatility já possui perfis construídos previamente. Contudo, para ambientes Linux, devido à grande variação na quantidade de versões de *kernel*s, sub-*kernel*s e *kernel*s customizados, onde cada compilação depende das opções de configuração de compilação, os perfis podem variar drasticamente. Com isso, é necessária a construção de um perfil específico para cada *kernel* a ser analisado.

Conforme [LIGH et al. 2014], para construção do perfil é necessário utilizar o compilador de código nativo, os arquivos de cabeçalho do *kernel* alvo, e informações das estruturas de dados utilizadas no *kernel*. A informações dessas estruturas podem ser extraídas através de uma ferramenta, denominada *dwarfdump*, que percorre informações de debug de arquivos ELF, como os do *kernel* Linux e módulos de *kernel*, e extrai informações de debuging no formato DWARF[16]. Essa ferramenta, dentre outras coisas, é capaz de fornecer definições das estruturas de dados internas do *kernel* necessárias para construção do perfil. Essa ferramenta pode ser baixada pelos canais de pacotes das estações de análise como o pacote `dwarfdump` nos sistemas Ubuntu/Debian ou `libdwarf-tools`, no Fedora.

Para automatizar a construção do perfil, o *framework* inclui um arquivo `Makefile`, disponível no subdiretório da instalação `[volatilityhome]/tools/Linux`, no qual é

[15] O ambiente Python pode ser baixado do site http://www.python.org.

[16] DWARF é um formato de arquivo de debugging utilizado por vários compiladores e ferramentas de debug, que permite debug no nível do código-fonte. Informações do formato disponíveis em http://dwarfstd.org

necessária a edição dos caminhos para o código fonte do *kernel*, compilador cruzado e da ferramenta DWARF, como ilustrado na Figura 3.3.

```
obj-m += module.o
KDIR := ~/kernel-source
CCPATH      :=      /path/to/android-ndk/toolchains/arm-linux-androideabi-
4.6/prebuilt/linux-x86/bin/
DWARFDUMP := /path/to/dwarf-dir/dwarfdump/dwarfdump
-include version.mk
all: dwarf
dwarf: module.c
$(MAKE) ARCH=arm CROSS_COMPILE=$(CCPATH)/arm-linux-androideabi- -C $(KDIR)
CONFIG_DEBUG_INFO=y M=$(PWD) modules
$(DWARFDUMP) -di module.ko > module.dwarf
```

Figura 3.3 – Comandos para compilação do módulo DWARF

Em seguida, é necessário executar o arquivo `Makefile` editado e depois combinar o arquivo produto da ferramenta DWARF (`module.dwarf`) com o arquivo `System.map` do código fonte do *kernel* alvo em um arquivo compactado (zip) como ilustrado na Figura 3.4.

```
$ make

$ zip /path/to/volatility/plugins/overlays/linux/LinuxExemplo_3.4.67.zip
/path/to/module.dwarf
/path/to/System.map
```

Figura 3.4 – Comandos para empacotamento do perfil para uso no Volatility

Esse arquivo pode ser posto no diretório denominado `[volatility home]/volatility/plugins/overlays/linux/` ou o caminho completo do arquivo passado como parâmetro na invocação das extensões do *framework*.

3.2.2. USO DO FRAMEWORK VOLATILITY

Para uso do *framework* com o perfil construído, o arquivo de perfil pode ser designado como parâmetro na chamada das extensões do *framework* (`--profile`). O nome obedece a nomenclatura: `Linux+NomedoArquivoZip+ARM`, se for um perfil para arquitetura ARM. Outra opção, é copiar o arquivo para o diretório de *overlay* em `[volatilityhome]/volatility/plugins/overlays/linux`, evitando-se ter que passar o caminho completo na designação do perfil.

Dessa forma, como exemplo, podemos executar uma extensão para Linux para recuperar a lista de processos ativos com o seguinte comando ilustrado na Figura 3.5.

```
$ python vol.py --profile=LinuxExemplo_3_4_67 -f
/path/to/memory/extracao.lime linux_pslist
```

Figura 3.5 – Exemplo de uso do perfil criado

Sendo o parâmetro `-f` o caminho para o arquivo contendo a extração, e `linux_pslist` o nome da extensão invocada.

3.2.3. VTYPES

Os *vtypes* são as definições de estrutura e *parsing* utilizadas pelo *framework* Volatility. Um profile construído para um determinado *kernel* contém um conjunto de *vtypes* para interpretação específica daquele ambiente possibilitando a recuperação de diversos tipos de dados da memória.

É possível estender essas definições de estruturas, com uso de *overlays*, pois infelizmente os símbolos de debug gerados para um perfil não contém informações suficientes para que o *framework* gere todos os tipos automaticamente, e geralmente se precisa recuperar, ou referenciar ponteiros para estruturas que não se sabe a natureza por serem do tipo *void*. O tipo de objeto pode ser conhecido através de engenharia reversa ou tentativa e erro.

Nesse trabalho, várias das estruturas foram levantadas através da interpretação do código fonte AOSP ou por tentativa e erro, e estão listadas em anexo no arquivo `ART_vtype.py`.

3.2.4. ALTERAÇÃO DO VOLATILITY PARA INTERPRETAÇÃO DO ANDROID

Para o correto funcionamento da extensão *proc_maps*, destinada a extrair mapeamentos de memória de um processo, foi necessário modificar o código original do arquivo de *overlay* no diretório de instalação do *framework* denominado `volatility/plugins/overlays/linux/linux.py`. No código fonte desse arquivo, na classe listada abaixo (`vm_area_struct`), foi necessário acrescentar a condição assinalada para ignorar o recurso de *Virtual Dynamic Shared Object* (vDSO) não existente no *kernel* Android dos dispositivos analisados, conforme ilustrado na Figura 3.6.

```
class vm_area_struct(obj.CType):
    def vm_name(self, task):
        if self.vm_file:
            fname = linux_common.get_path(task, self.vm_file)
        elif  self.vm_start  <=  task.mm.start_brk  and  self.vm_end  >=
task.mm.brk:
            fname = "[heap]"
        elif  self.vm_start  <=  task.mm.start_stack  and  self.vm_end  >=
task.mm.start_stack:
            fname = "[stack]"
        elif  hasattr(self.vm_mm.context,  "vdso")  and  self.vm_start  ==
self.vm_mm.context.vdso:
            fname = "[vdso]"
        else:
            fname = "Anonymous Mapping"

        return fname
```

Figura 3.6 – Alteração do Volatility (assinalada) para funcionamento com os *kernels* utilizados.

3.3. ANÁLISE DE DADOS DO KERNEL LINUX

Com uma aquisição de memória e o ambiente de análise preparado, é possível, através das extensões do Volatility padrões para ambiente Linux, obter a extração de diversas informações úteis para investigação. Em [HILGERS et al. 2014] foram apresentadas ferramentas que permitem a recuperação de dados de aplicações no ambiente Linux e dados de objetos da máquina virtual nas versões do Android sem o ART.

```
>python vol.py --profile=LinuxLinuxGoldfish_3_4_67ARM -f memdumpWhatsAppChat.lime  -p 1206
linux_proc_maps
Volatility Foundation Volatility Framework 2.4
Pid      Start              End                Flags       Pgoff Major  Minor  Inode
File Path
-------- ------------------ ------------------ ------ ---------- ------ ------ ---------- ---
------
    1206 0x0000000012c00000 0x0000000012e01000 rw-           0x0      0      4       2003
/dev/ashmem/dalvik-main space
    1206 0x0000000012e01000 0x0000000013c91000 rw-      0x201000      0      4       2003
/dev/ashmem/dalvik-main space
    1206 0x0000000013c91000 0x0000000013e81000 ---     0x1091000      0      4       2003
/dev/ashmem/dalvik-main space
    1206 0x0000000013e81000 0x0000000016c00000 rw-     0x1281000      0      4       2003
/dev/ashmem/dalvik-main space
    1206 0x0000000016c00000 0x000000001ac00000 r--           0x0      0      4       2004
/dev/ashmem/dalvik-main space
    1206 0x00000000700c7000 0x0000000070be8000 rw-           0x0     31      1       7053
/data/dalvik-cache/arm/system@framework@boot.art
    1206 0x0000000070be8000 0x00000000726a7000 r--           0x0     31      1       7054
/data/dalvik-cache/arm/system@framework@boot.oat
    1206 0x00000000726a7000 0x0000000073b5f000 r-x     0x1abf000     31      1       7054
/data/dalvik-cache/arm/system@framework@boot.oat
    1206 0x0000000073b5f000 0x0000000073b60000 rw-     0x2f77000     31      1       7054
/data/dalvik-cache/arm/system@framework@boot.oat
    1206 0x0000000073b60000 0x0000000073cc3000 rw-           0x0      0      4       2002
/dev/ashmem/dalvik-non moving space
    1206 0x0000000073cc3000 0x0000000073cc4000 rw-           0x0      0      4       2482
/dev/ashmem/dalvik-alloc space
    1206 0x0000000073cc4000 0x0000000073efe000 rw-        0x1000      0      4       2482 …
```

Figura 3.7 - Trecho da execução de ferramenta para Linux sobre uma extração de memória de um dispositivo Android emulado.

Várias informações podem ser coletadas através do *framework* Volatility, dentre elas a listagem do mapeamento de memória de um processo, através da extensão `linux_proc_maps`, que inclui o endereço virtual de cada item mapeado e *flags* de acesso da *heap*, pilha e bibliotecas de linkagem dinâmica mapeadas. Como ART trabalha intensamente com mapeamento de memória, essa extensão é muito utilizada internamente pelas ferramentas utilizadas na técnica desenvolvida nesse trabalho. Na Figura 3.7, é ilustrado um trecho da execução da extensão sobre uma extração de memória de um dispositivo emulado.

3.4. ANÁLISE DE DADOS DE OBJETOS DA MÁQUINA VIRTUAL

Com a aquisição de memória RAM efetuada e as informações sobre o ambiente de execução levantadas, é possível executar o procedimento que percorre a área de memória da *heap* destinada a alocação de objetos menores que 12 KiB e recuperar dos agrupamentos de páginas os *slots* com os dados de cada objeto alocado, ou mesmo dados de *slots* desalocados recuperáveis. Assim, conforme esquema da Figura 3.8 (exemplificado para uma *heap* de 64 MiB), com os bytes do *slot* que contém dados de um determinado objeto e com leiaute de memória para a classe (definido em função do endereço da classe presente nos bytes iniciais, aqui denominado `klass`), é possível recuperar as informações do objeto.

Conforme descrito para a técnica proposta, a recuperação de todos os *slots* com dados de objetos Java alocados na *heap* é através da varredura de todo o mapeamento de páginas mantidas pelo alocador *RosAlloc* (arquivo mapeado em `/dev/ashmem/dalvik-rosalloc page map`), percorrendo os agrupamentos nas páginas do tipo *kPageMapRun* e *kPageMapRunPart* que são destinadas ao armazenamento de dados dos objetos menores que 12KiB, e a partir dos dados obtidos, realizar a interpretação dos dados de acordo com o tipo de classe definida para o objeto. Esse arquivo de mapeamento (`RosAlloc PageMap`) é uma sequência de bytes correspondente a uma página (no exemplo, páginas com 4KiB de tamanho) da *heap* onde é armazenado o tipo de página conforme codificação presente na Tabela 2.4.

Como mencionado para descrição da técnica, as ferramentas de software construídas, na forma de arquivos de código Python (códigos fonte em anexo), foram utilizadas para o processo de recuperação dos dados dos *slots* e para interpretação desses dados como objetos Java.

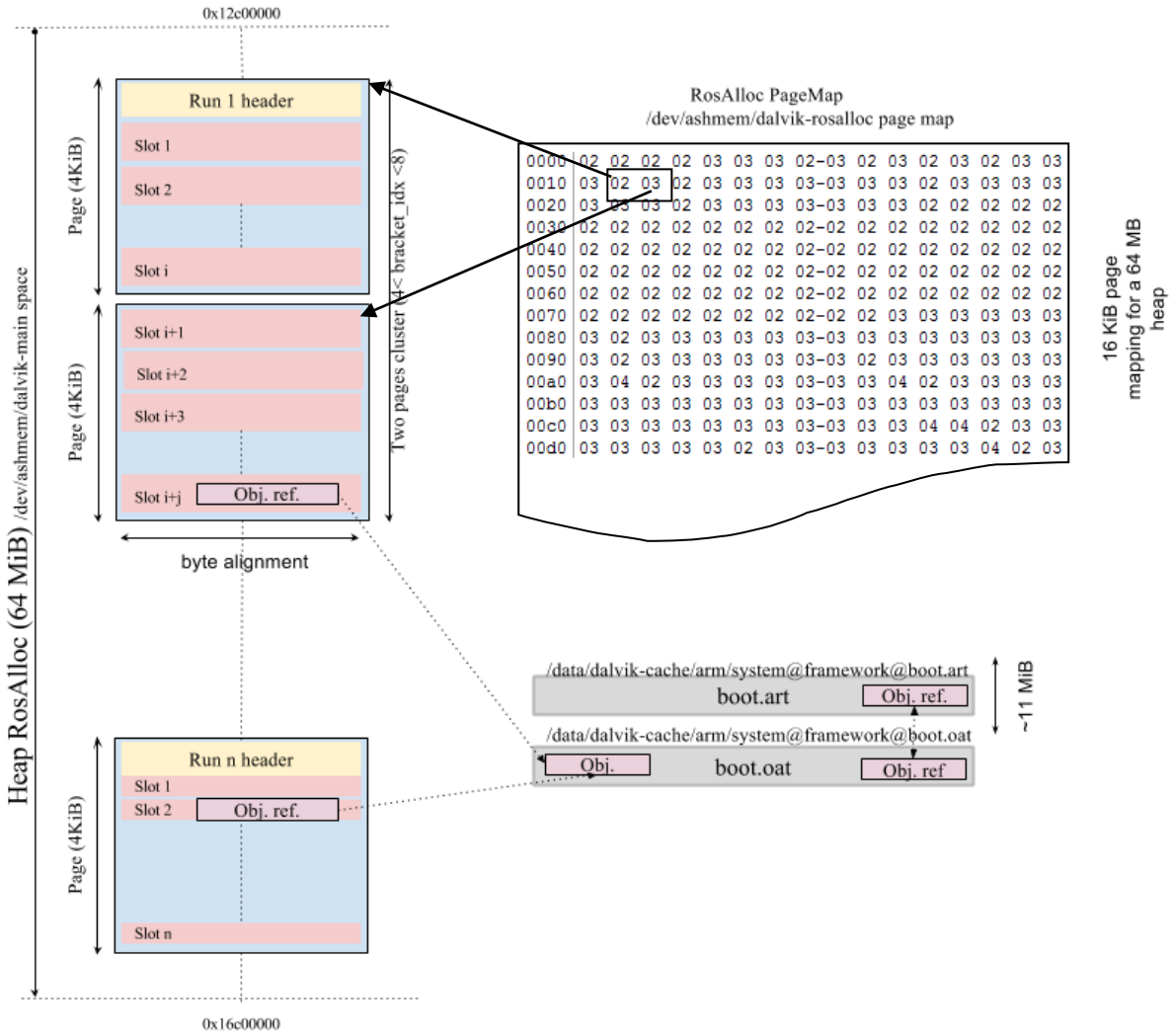


Figura 3.8 - Mapeamento de uma heap (64MiB) pelo alocador RosAlloc

Para interpretação dos dados, foi criado *vtypes* para o *framework* Volatility (códigos-fonte listados no Apêndice C) que possibilitam interpretar classes de objetos e estruturas utilizadas pelo ambiente de execução para armazenamento de propriedades de sistema, estruturas de arquivos DEX, cabeçalhos OAT, ART e páginas do tipo *kPageMapRun*. Essas definições, descritas nas Tabelas 2.2 a 2.7, estão contidas no arquivo denominado `ART_vtypes.py`, e foram acrescentadas no diretório do *framework* Volatility denominado `[VolatilityHome]/volatility/plugins/overlays/linux/`.

Tabela 3.1 - Mapeamentos utilizados na recuperação de dados de arquivos mapeados

Nome do VType para arquivos mapeados	**Descrição**
ArtHeader	Cabeçalho de um arquivo ART (V. Tabela 3.2)
OatHeader	Cabeçaho de um arquivo OAT (V. Tabela 3.3)
OatDexHeader	Cabeçalho de um arquivo DEX dentro de OAT (V. Figura 3.2)

DexHeader	Cabeçalho de um arquivo DEX (V. Tabela Figura 3.2)
DexMapItem	Item do mapa de um DEX
DexMap	Mapa de um DEX
OatClassHeader	Cabeçalho de uma classe em um OAT

Fonte: Código-fonte em [AOSP 2014]

Tabela 3.2- Mapeamentos de classes utilizados na interpretação dos dados de objetos recuperados

Nome do VType para classes mapeadas	Descrição
Ljava/lang/Object;	Ancestral de todas as classes.
Ljava/lang/Class;	Meta-classe das classes.
Ljava/lang/DexCache;	Contém dados referentes ao DEX associado a classe.
Ljava/lang/reflect/ArtField;	Utilizada internamente pelo ART para acesso aos campos de uma classe.
Ljava/lang/reflect/ArtMethod;	Utilizada internamente pelo ART para acesso aos métodos de uma classe.
Ljava/lang/reflect/Proxy;	Relacionada a implementações dinâmicas de interfaces em tempo de execução.
Ljava/lang/ref/Reference;	Relacioanda a guarda de referencia a objetos. Utilizada pelo sistema de GC.
Ljava/lang/StackTraceElement;	Relacionada a composição de uma pilha gerada pelo controle de exceção.
Ljava/lang/Throwable;	Superclasse relacionada com todos os erros e exceções na execução do código Java.
Ljava/lang/String;	Relacionada a composição de cadeia de caracteres. Geralmente codificadas em UTF-16.
[L	Classificador para todas classes organizadas em forma de array.
[Ljava/lang/Object;	Array de classes do tipo Objeto.
[Ljava/lang/lang/reflect/ArtField;	Array de classes do tipo ArtField.

Fonte: Código-fonte em [AOSP 2014]

Tabela 3.3 - Mapeamento do cabeçalho de página utilizado pelo alocador RosAlloc

Nome do VType para o RosAlloc	Descrição
rosalloc/runfixedhead	Parte fixa do cabeçalho das páginas que compõem os agrupamentos de slots (*brackets*).

Fonte: Código-fonte em [AOSP 2014]

Tabela 3.4 - Estruturas relacionadas ao armazenamento de propriedades de sistema em estrutura de dados do tipo radix tree

Nome do VType para mapeamento da estrutura de propriedades de sistema (em *radix tree*)	**Descrição**
prop_area	Raiz da radix tree
prop_bt	Dados de um nodo
prop_info	Informação associada a um determinado nodo

Fonte: Código-fonte em [AOSP 2014]

Para a recuperação dos dados, foram construídas as seguintes extensões que possibilitam, utilizando os *vtypes* criados, a recuperação de dados relacionados ao ambiente de execução:

- `art_extract_properties_data` - extrai as propriedades gerenciadas pelo serviço `System Properties`. Como argumento é necessário o ID do processo. Através dessa ferramenta é possível ter acesso a propriedade `dalvik.vm.heapsize` que tem armazenada o valor, em MiB, do tamanho da *heap*. Esse valor pode ser utilizado como parâmetro no processo de extração de objetos.
- `art_find_oat` – localiza arquivos OAT mapeados para um processo de aplicação. Como argumento é passado o ID do processo (PID). Os endereços virtuais do processo alvo, contendo mapeamentos de arquivos OAT, podem ser utilizados para extração de dados referentes às classes contidas no código DEX embutido no OAT.
- `art_dump_rosalloc_heap_objects` - percorre uma região de *heap,* utilizando o mapa de bits das páginas de alocação, extraindo dados de todos os objetos ou de um objeto de uma classe específica identificada por índices do DEX, endereço da classe ou sequência de bytes específica. Como parâmetros são necessários o ID do processo alvo (PID), o endereço onde o arquivo de imagem está mapeado (BOOT_ART_ADDRESS) para possibilitar a interpretação dos tipos raiz do *framework* Java a partir dos endereços contidos no cabeçalho, o endereço onde a *heap* de objetos está mapeada (ROSALLOC_HEAP_ADDRESS), o endereço em onde está o mapeamento de páginas da *heap* e o tamanho da *heap*. É possível a busca de uma sequência de bytes que possa existir em *slot*.
- `art_extract_object_data` - recupera dados de um objeto baseado no endereço e nas informações de classes mapeadas, contornando casos onde dados do objeto

alvo não estão presentes na memória por ausência da página. Como parâmetros são necessários: o ID do processo alvo (PID), o endereço onde o arquivo de imagem está mapeado (BOOT_ART_ADDRESS), o endereço onde estão localizados os dados do objeto (OBJECT_ADDRESS) e, opcionalmente, o nome da classe (CLASS_ADDRESS) para interpretação forçada de uma classe ou ponteiro.

- `art_extract_image_data` - recupera o cabeçalho do arquivo de imagem ART. Como parâmetro pode ser utilizado o ID do processo alvo (PID), ou nenhum parâmetro, recuperando dados do próprio processo Zygote. Dentre as informações apresentadas, a localização da imagem do *framework* e os endereços das classes raízes são essenciais para interpretação dos dados de objetos recuperados.
- `art_extract_oat_data` - decodifica um arquivo OAT mapeado em memória, extraíndo informações do primeiro DEX embutido, incluindo extração do arquivo DEX que pode ser decompilado por ferramentas específicas para análise estática de dados. Muitas vezes a região de memória do arquivo DEX possui páginas vazias não carregadas pelo gerenciamento de memória, o que dificulta a interpretação e a engenharia reversa do código.
- `art_dex_data` – auxilia na extração de dados referentes a `string`, `types`, `proto`, `fields`, `method` e `class lists` do arquivo DEX embutido em um mapeamento de memória referente a um OAT. Durante a execução da extensão `art_extract_oat_data`, são gerados arquivos contendo listas dos dados extraídos que podem ser utilizados por outras ferramentas, visando acelerar o tempo de extração de dados. Um dos parâmetros é o endereço inicial do arquivo OAT (`offset_oat_data_begin`). Essas informações extraídas do arquivo DEX podem ser utilizadas para decodificação do objeto, como o índice do tipo e da definição da classe.
- `art_extract_oat_dex_data` - decodifica um arquivo OAT mapeado em memória, extraíndo os binários dos DEXs que pode ser decompilado por ferramentas específicas para análise estática de dados. Muitas vezes a região de memória do arquivo DEX possui páginas vazias não carregadas pelo gerenciamento de memória, tornando o arquivo não passível de decompilação por ferramentas tradicionais.

4. VALIDAÇÃO EXPERIMENTAL

A validação experimental da técnica proposta foi realizada em um dispositivo emulado e em um dispositivo real, ambos operando com o ART. Uma aquisição total da memória RAM de cada dispositivo foi realizada utilizando a técnica descrita em [SYLVE et al. 2012] enquanto estavam com aplicações ativas, incluindo uma aplicação de bate-papo (WhatsApp v 2.12.510)[17].

Para a aquisição de memória dos dispositivos, foi necessário utilizar um acesso de usuário com privilégios elevados (root) para realizar a substituição do kernel original por um configurado para permitir o carregamento de módulos kernel sem validações de segurança.

O usuário com privilégio elevado root é disponibilizado por padrão no ambiente emulado, enquanto que para o aparelho real foi obtido o privilégio elevado através da ferramenta Kingo [KINGO 2016].

O código-fonte dos respectivos kernels foram compilados de acordo com as instruções disponíveis no site do AOSP. A estação utilizada para o processo de compilação-cruzada e análise consistia do Santoku Linux versão 0.4, com a instalação do Android NDK (Release 8e) e do *framework* Volatility (versão 2.4) como descrito nos respectivos sites dos projetos. A configuração do ambiente foi baseada na construção experimental utilizada em [HØGSET 2015] e durante cada aquisição de memória, foi utilizada a transferência direta dos dados por TCP do dispositivo para a estação de análise.

4.1. VALIDAÇÃO COM DISPOSITIVO EMULADO

O dispositivo emulado era composto de uma Android Virtual Device (AVD) configurado com os parâmetros CPU/ABI ARM (armeabi-v7a), 768 MB RAM, Target Android 5.0.1 (API level 21), build number “sdk_phone_armv7-eng 5.0.2 LSY64 1772600 test-keys”, hw.device.name Nexus 5 e vm.heapSize 64 MB.

4.1.1. CONFIGURAÇÃO DO AMBIENTE

O dispositivo alvo utilizado para aquisição de memória foi o emulador Android, disponível no pacote de ferramentas de desenvolvimento Android SDK Revision 23.0.2,

[17] WhatsApp é um aplicativo multiplataforma de mensagens instantâneas e chamadas de voz para smartphones disponível em https://www.whatsapp.com/.

com as aplicações padrão e com o aplicativo de bate-papo supracitado. O código fonte do kernel versão 3.4.67 disponível para o emulador (*goldfish*) foi obtido do site do AOSP.

4.2. APLICAÇÃO DA TÉCNICA

O dispositivo emulado foi inicializado com o a versão do *kernel* 3.4.67 com o aplicativo de bate-papo WhatsApp (versão 2.12.510) em execução onde previamente foram trocadas mensagens com um interlocutor utilizando um aparelho real e, em seguida, com o *Activity* da conversa realizada fora de foco (em *background*), foi realizada aquisição de dados, através do carregamento do módulo LiME e envio dos dados por TCP para estação de análise, conforme procedimento previamente descrito no item 3.1.

Para início da técnica, foi utilizada a extensão *linux_pslist* para do *framework* Volatility para recuperar a tabela de processos em execução. Dessa forma, é possível localizar o identificador do processo (PID) alvo da análise, conforme ilustrado na Figura 4.1.

```
>python vol.py --profile=LinuxLinuxGoldfish_3_4_67ARM -f
memdumpWhatsAppChat.lime linux_pslist
Offset     Name           Pid     Uid        Gid    DTB         Start Time
---------- ------------- ------- ---------- ------ ---------- ----------
0xee01cc00 init             1         0       0        0x2d90c000 2016-03-16
12:39:36 UTC+0000
…
0xd3b09400  com.whatsapp    1206       10054  10054    0x10824000  2016-03-16
12:50:44 UTC+0000
....
```

Figura 4.1- Execução da ferramenta `linux_pslist`

Identificada a aplicação alvo, PID 1206 (*com.whatsapp*), foi possível através da ferramenta `art_extract_properties_data`, extrair dados das propriedades do ambiente e, dentre eles, o tamanho da *heap* de objetos Java, utilizado posteriormente como parâmetro na recuperação de objetos alvo, conforme Figura 4.2.

```
>python vol.py --profile=LinuxLinuxGoldfish_3_4_67ARM -f
memdumpWhatsAppChat.lime art_extract_properties_data –p 1206
...
[ro.boot.hardware]= [goldfish]
...
[dalvik.vm.heapsize]= [64m]
...
[persist.sys.dalvik.vm.lib.2]= [libart.so]
...
[persist.sys.timezone]= [GMT]
...
```

Figura 4.2 – Execução da ferramenta `art_extract_properties_data`

Em seguida, utilizando a ferramenta `art_extract_image_data` com o identificador do processo PID como parâmetro, foi possível recuperar os dados referentes ao ambiente de execução do processo alvo, como o endereçamento do mapeamento do *framework*, conforme ilustrado na Figura 4.3.

```
>python vol.py --profile=LinuxLinuxGoldfish_3_4_67ARM -f
memdumpWhatsAppChat.lime art_extract_image_data -p 1206
com.whatsapp
Boot.art
-----------------------
initial_offset:0x700c7000
end_offset:0x70be8000
OatHeader
-----------------------
image_begin:0x700c7000
…
oat_checksum:0xbd5a21c9L
oat_file_begin:0x70be8000
oat_data_begin:0x70be9000
…
image_roots:0x70bb8840
…
   kClassRoots:0x70bb8948
        0x1 Ljava/lang/Class; 0x700c7220L
        0x2 Ljava/lang/Object; 0x700f7240L
…
        0x5 Ljava/lang/String; 0x700df8f0L
        0x6 Ljava/lang/DexCache; 0x700c74f0L
…
        0x8 Ljava/lang/reflect/ArtField; 0x700f7640L
…
…       0xc [Ljava/lang/reflect/ArtField; 0x700f74a0L
        0x1d [C 0x700f6fd8L
…
```

Figura 4.3 - Execução da ferramenta `art_extract_image_data`

Identificaram-se diversas informações do cabeçalho, dentre elas, o *offset* na memória com a localização do mapeamento do *framework* (0x700c7000), que serve de base para recuperação de informações de diversas classes raízes (`kClassRoots`), como o endereço da definição da classe `java.lang.String` (0x700df8f0) que possibilita interpretar os dados de vários objetos, inclusive de textos armazenados. Também é possível identificar nos dados recuperados, o endereço referente ao OAT com objetos do *framework* Android (0x70be80000).

Com as informações colhidas, foi possível também a recuperação de dados referentes aos arquivos OAT (além do OAT do *framework*, não analisado nesse processo) utilizados pelo processo alvo através da ferramenta `art_find_oat`, incluindo o endereço dos arquivos que possibilitam recuperar a localização dos arquivos DEX, conforme Figura 4.4.

```
>python vol.py --profile=LinuxLinuxGoldfish_3_4_67ARM -f
memdumpWhatsAppChat.lime art_find_oat -p 1206
oat                                                          offset_
------------------------------------------------------------ --------
/data/dalvik-cache/arm/system@app@webview@webview.apk@classes.dex   0xa06dc000L
/data/dalvik-cache/arm/data@app@com.whatsapp-1@base.apk@classes.dex 0xa5a74000L
```

Figura 4.4 - Execução da ferramenta art_find_oat

O primeiro OAT listado é referente ao código do componente webview, relacionado ao recurso de navegação WEB por uma aplicação. O segundo OAT, referente à aplicação alvo (com.whatsapp). Através da ferramenta art_extract_oat_dex foi possível extrair os arquivos DEX (no formato ENDERECO_OAT.PID.dex.##.OFFSET_DEX). Algumas páginas de memórias desse mapeamento podem não estar disponíveis na memória, o que resultaria em um arquivo inconsistente que dificulta a análise estática do código com auxílio de ferramentas tradicionais de decompilação. Conforme Figura 4.5, foi executado o comando para extração de informações do OAT mapeado para o aplicativo alvo.

```
python vol.py --profile=LinuxLinuxGoldfish_3_4_67ARM -f
memdumpWhatsAppChat.lime  -p 1206 art_extract_oat_dex -o 0xa5a74000 -D .
oat
-----------------------
initial_offset:0xa5a75000L
OatHeader
-----------------------
instruction_set:0x3
instruction_set_features:0x0
dex_file_count:0x1
executable_offset:0xbcf000
interpreter_to_interpreter_bridge_offset:0x0
interpreter_to_compiled_code_bridge_offset:0x0
jni_dlsym_lookup_offset_:0x0
portable_imt_conflict_trampoline_offset:0x0
portable_resolution_trampoline_offset:0x0
portable_to_interpreter_bridge_offset:0x0
quick_generic_jni_trampoline_offset:0x0
quick_imt_conflict_trampoline_offset:0x0
quick_resolution_trampoline_offset:0x0
quick_to_interpreter_bridge_offset:0x0
image_patch_delta:0xc7000
image_file_location_oat_checksum:0xbd5a21c9L
image_file_location_oat_data_begin:0x70be9000
key_value_store_size:0x161
key_value_store:dex2oat-cmdline --zip-fd=6 --zip-
location=/data/app/com.whatsapp-1/base.apk --oat-fd=7 --oat-
location=/data/dalvik-cache/arm/data@app@com.whatsapp-1@base.apk@classe
s.dex --instruction-set=arm --instruction-set-features=default --runtime-
arg -Xms64m --runtime-arg -Xmx512m dex2oat-host Arm image-location
/data/dalvik-cache/arm/system@framework@
boot.art
Dex Pid   OAT              Start           Size            Output File
--- ----- -------------- --------------- --------------- ----------------
  1 0...6 0xa5a74000L     0xa5a7ba08L     0x8e2f80L
.\0xa5a74000L.1206.dex.1.0xa5a7ba08
```

Figura 4.5 - Execução da ferramenta art_extract_oat_dex

O arquivo DEX extraído (**`0xa5a74000L.1206.dex.1.0xa5a7ba08`**) pode ser carregado em ferramentas de decompilação por estar íntegro (sem falta de dados de possíveis páginas não carregadas em memória), sendo possível inspecioná-lo utilizando engenharia reversa.

Também foi possível utilizar a ferramenta `art_extract_oat_data`, que faz a extração dos dados referentes a lista de strings, protótipos, tipos e classes do arquivo DEX para arquivos que permitem tanto a pesquisa direta de símbolos (identificando índices de elementos, como classes). Os arquivos gerados puderam ser utilizados na ferramenta `art_extract_object_data`, responsável pela interpretação de dados de objeto em determinado endereço fornecido. Para isso, foi necessário incluir o endereço do OAT que permitiu a interpretação dos símbolos referente aos campos internos de uma classe de objeto específica. Os arquivos extraídos têm o formato `ENDEREÇO_OAT_pid##_dex.[string|type|proto|field|method|class].list`, conforme ilustrado na Figura 4.6, executado para o OAT em 0xA5A74000.

```
0xa5a74000L_pid1206_dex.class.list
0xa5a74000L_pid1206_dex.field.list
0xa5a74000L_pid1206_dex.method.list
0xa5a74000L_pid1206_dex.proto.list
0xa5a74000L_pid1206_dex.string.list
0xa5a74000L_pid1206_dex.type.list
```

Figura 4.6 – Listagem de arquivos gerados pela ferramenta `art_extract_oat_data`

Pode-se inspecionar o arquivo contendo a lista de classes (`0xa5a74000L_pid1206_dex.class.list`), onde foi possível identificar o índice 5012 (0x1394) para uma classe específica (`Lcom/whatsapp/protocol/l`). Após engenharia reversa, a classe aparentou estar ligada ao armazenamento de conversas que são apresentadas ativas no aplicativo alvo, conforme Figura 4.7.

```
sI5012
S'Lcom/whatsapp/protocol/l;'
```

Figura 4.7 – Trecho contendo listagem de classes do DEX contido no OAT examinado

Observa-se que caso o APK da aplicação alvo esteja disponível, também é possível realizar diretamente a análise de alguns componentes dos arquivos DEX diretamente de um arquivo APK. Na Figura 4.8, segue um trecho da decompilação realizada através da ferramenta `dexdump` (disponível no próprio Android SDK) sobre o APK da versão utilizada no dispositivo emulado, confirmando a eficácia da ferramenta desenvolvida para técnica.

```
Class #5012            -
  Class descriptor  : 'Lcom/whatsapp/protocol/l;'
  Access flags      : 0x0001 (PUBLIC)
  Superclass        : 'Ljava/lang/Object;'
  …
  Instance fields   -
    #0              : (in Lcom/whatsapp/protocol/l;)
      name          : 'A'
      type          : 'Ljava/lang/String;'
      access        : 0x0001 (PUBLIC)
    #1              : (in Lcom/whatsapp/protocol/l;)
      name          : 'B'
      type          : 'Ljava/lang/Integer;'
      access        : 0x0001 (PUBLIC)
    #2              : (in Lcom/whatsapp/protocol/l;)
      name          : 'C'
      type          : 'Z'
      access        : 0x0001 (PUBLIC)
    #3              : (in Lcom/whatsapp/protocol/l;)
      name          : 'E'
      type          : 'I'
      access        : 0x0001 (PUBLIC)
    #4              : (in Lcom/whatsapp/protocol/l;)
      name          : 'G'
      type          : 'I'
      access        : 0x0001 (PUBLIC)
    #5              : (in Lcom/whatsapp/protocol/l;)
      name          : 'H'
      type          : 'Lcom/whatsapp/protocol/i;'
      access        : 0x0001 (PUBLIC)
…
```

Figura 4.8 – Trecho contendo detalhes da decompilação do DEX diretamente do APK examinado

Pelas três análises feitas sobre o OAT da aplicação alvo, pode se recuperar do arquivo DEX os índices identificadores de uma classe. Esse identificador (0x1394) pode ser buscado na lista de referências de objetos da *heap*, através do levantamento de objetos que são implementações dessas classes.

Para construção da lista de referências foram utilizados, além dos parâmetros já utilizados nas demais ferramentas (pid e endereço da imagem), os seguintes:

- ROSALLOC_HEAP_ADDRESS, endereço da *heap* (0x12c00000), localizado como o primeiro endereço mapeado e identificado através da extensão `linux_proc_maps` para o arquivo `/dev/ashmem/dalvik-main space`;
- ROSALLOC_PAGEMAP_ADDRESS, endereço do mapeamento de páginas do RosAlloc (0xb1d70000), localizado como o primeiro endereço mapeado e identificado através da extensão `linux_proc_maps` para o arquivo `/dev/ashmem/rosalloc page map`;

- ROSALLOC_PAGEMAP_SIZE, tamanho da *heap* (0x4000000), determinado na propriedade de sistema (`dalvik.vm.heapsize`) recuperada.

Na saída extensa produzida pela ferramenta, ilustrada na Figura 4.9, pode ser localizado o endereço da classe cuja a meta-classe ancestral possui o identificador (`dexClassDefIndex`) para classe alvo.

```
>python vol.py --profile=LinuxLinuxGoldfish_3_4_67ARM -f memdumpWhatsAppChat.lime -p 1206 -b 0x700c7000
art_dump_rosalloc_heap_objects -e 0x12c00000 -m 0xb1d70000 -s 0x4000000
address     page bracket slot obj class                          data
----------- ---- ------- ---- ------------------------------- -----
0x12c19020L 19 29 50 Ljava/lang/Class;(0x700c7220)   dexCache:0x12c01610L name: dexClassDefIndex:0x1394L
iFields:0x12c04900L
```

Figura 4.9 – Trecho da execução da ferramenta `art_dump_rosalloc_heap_objects` **na busca do endereço da classe ancestral**

Com isso, foi possível determinar (com caminho semelhante ao processo implementado na ferramenta baseado na programação por reflexão) que objetos da classe identificada pela classe descrita no endereço 0x12c19020 eram objetos relacionados com armazenamento de mensagens de bate-papos ativos do aplicativo alvo, e que no endereço 0x1204900 estavam descritos os campos internos da classe. Através de uma busca no próprio *dump* já realizado, foi possível identificar quatro objetos que eram instâncias dessa classe:, conforme ilustrado na Figura 4.10.

```
>python vol.py --profile=LinuxLinuxGoldfish_3_4_67ARM -f
memdumpWhatsAppChat.lime -p 1206 -b 0x700c7000
art_dump_rosalloc_heap_objects -e 0x12c00000 -m 0xb1d70000 -s 0x4000000
…
address        page    bracket slot obj class                          data
-------------- ------- ------- ---- ------------------------------- --------------------------------
0x1384d2c0L       3149      13    2 *(FOUND)* 0x12c19020             20 90 c1 12 51 4d e3 8c 00 00 00
0x1384e0c0L       3149      13   18 *(FOUND)* 0x12c19020             20 90 c1 12 49 3c 07 bf 00 00 00
0x1384f240L       3149      13   38 *(FOUND)* 0x12c19020             20 90 c1 12 fb 2f 6d b4 00 00 00
0x13850820L       3149      13   63 *(FOUND)* 0x12c19020             20 90 c1 12 25 46 dc bb 00 00 00
```

Figura 4.10 - Trecho da execução da ferramenta `art_dump_rosalloc_heap_objects` **na busca de objetos de uma determinada classe**

Para o primeiro objeto localizado em 0x1384d2c0, através da ferramenta `art_extract_object_data`, podemos recuperar as seguintes informações, conforme ilustrado na Figura 4.11.

```
>python vol.py --profile=LinuxLinuxGoldfish_3_4_67ARM -f
memdumpWhatsAppChat.lime  -p 1206 -b 0x700c7000 art_extract_object_data -o
0x1384d2c0
Object Address: 0x1384d2c0L
Address for IMAGE BOOT: 0x700c7000L
target art class 0x12c19020L (classtype not root)
```

Figura 4.11 - Execução da ferramenta `art_extract_object_data` **para recuperação de dados do do objeto analisado**

Esta é uma classe específica do aplicativo de bate-papo e não foi previamente mapeada para permitir a interpretação dos dados diretamente. Com isso, ascendemos na hierarquia utilizando a mesma ferramenta para obter a descrição da classe descrita em 0x12c19020, conforme ilustrado na Figura 4.12.

```
>python vol.py --profile=LinuxLinuxGoldfish_3_4_67ARM -f
memdumpWhatsAppChat.lime  -p 1206 -b 0x700c7000 art_extract_object_data -o
0x12c19020

Object Address: 0x12c19020L
Address for IMAGE BOOT: 0x700c7000L
target_art_class 0x700c7220L (classtype is root)
Class name: Ljava/lang/Class;
classLoader 0x12c02b20L
componentType 0x0L
dexCache 0x12c01610L Ljava/lang/DexCache;
directMethods 0x133ff980L [Ljava/lang/reflect/ArtMethod;
iFields 0x12c04900L [Ljava/lang/reflect/ArtField;
ifTable 0x0L None
imTable 0x0L None
name 0x0L None
sFields 0x13407500L [Ljava/lang/reflect/ArtField;
superClass 0x700f7240L Ljava/lang/Class;
verifyErrorClass 0x0L None
virtualMethods 0x133ff9c0L [Ljava/lang/reflect/ArtMethod;
vtable 0x0L None
accessFlags 0x80001L
classSize 0x1d4L
clinitThreadId 0x539L
dexClassDefIndex 0x1394L
dexTypeIndex 0x1810L
numReferenceInstanceFields 0x14L
numReferenceStaticFields 0x2L
objectSize 0xd8L
primitiveType 0x0L
referenceInstanceOffsets 0xbfffc00L
referenceStaticOffsets 0x3L
status 0xaL
```

Figura 4.12 - Execução da ferramenta `art_extract_object_data` **para recuperação de dados da classe do objeto analisado**

Dentre os dados recuperados, encontra-se o endereço referente ao *array* de propriedades da classe `java.lang.reflect.ArtField[]` em 0x12c04900L. Dessa forma, repetimos o uso da ferramenta para esse endereço e, como se trata de conjunto de campos de uma classe específica, é utilizado o endereço (0xA5a74000) do OAT, através do parâmetro `-x`, que contém o DEX da aplicação alvo. Com esse parâmetro opcional, as informações coletadas previamente pela ferramenta `art_extract_oat_data` são utilizadas para resolução dos nomes dos campos. Na Figura 4.13 é ilustrado um trecho da saída de execução da ferramenta com os dados recuperados.

```
>python vol.py --profile=LinuxLinuxGoldfish_3_4_67ARM -f
memdumpWhatsAppChat.lime -p 1206 -b 0x700c7000 art_extract_object_data -o
0x12c04900 -x 0xA5a74000

Object Address: 0x12c04900L
Address for IMAGE BOOT: 0x700c7000L
target_art_class 0x700f74a0L (classtype is root)
Class name: [Ljava/lang/reflect/ArtField;
0x12c04908L
Values: [43]
43
Dex strings
-----------------------
Loading 0xa5a74000L_pid1206_dex.string.list
Dex types
-----------------------
Loading 0xa5a74000L_pid1206_dex.type.list
Dex protos
-----------------------
Loading 0xa5a74000L_pid1206_dex.proto.list
Dex fields
-----------------------
Loading 0xa5a74000L_pid1206_dex.field.list
Dex methods
-----------------------
Loading 0xa5a74000L_pid1206_dex.method.list
Dex classes
-----------------------
Loading 0xa5a74000L_pid1206_dex.class.list
1 -> 0x73ce2c78L  declaringClass 0x12c19020L
 accessFlags 0x1L
 fieldDexIndex 0x53deL ('Lcom/whatsapp/protocol/l;', 'Ljava/lang/String;',
'A')
 offset 0x8L recover field data at (${base object address}+0x8L )
2 -> 0x73ce2c98L  declaringClass 0x12c19020L
 accessFlags 0x1L
 fieldDexIndex            0x53dfL            ('Lcom/whatsapp/protocol/l;',
'Ljava/lang/Integer;', 'B')
 offset 0xcL recover field data at (${base object address}+0xcL )
3 -> 0x73ce2d18L  declaringClass 0x12c19020L
 accessFlags 0x1L
 fieldDexIndex            0x53e5L            ('Lcom/whatsapp/protocol/l;',
'Lcom/whatsapp/protocol/i;', 'H')
 offset 0x10L recover field data at (${base object address}+0x10L )
4 -> 0x73ce2d38L  declaringClass 0x12c19020L
 accessFlags 0x2L
 fieldDexIndex 0x53e6L ('Lcom/whatsapp/protocol/l;', 'Ljava/lang/String;',
'I')
 offset 0x14L recover field data at (${base object address}+0x14L )
…
21 -> 0x73ce2d78L  declaringClass 0x12c19020L
 accessFlags 0x1L
 fieldDexIndex 0x53e8L ('Lcom/whatsapp/protocol/l;', 'J', 'K')
 offset 0x58L recover field data at (${base object address}+0x58L )
…
```

Figura 4.13 - Execução da ferra menta `art_extract_object_data` **para recuperação de dados da classe do objeto analisado com decodificação baseada nos dados da classe levantados na análise do arquivo OAT.**

No trecho ilustrado na Figura 4.13, produzido pela ferramenta com campos internos dessa classe, após análise reversa, pode-se identificar informações sobre horário, nome do interlocutor, estado de leitura, e outros. Dentre esses campos, foi identificado o campo denominado " I" (o nome desse identificador é fruto de um processo de obfuscação) na quarta posição da lista de campos. Nos dados extraídos pela ferramenta, é apresentado o *offset* nos dados do objeto para o endereço desse campo, identificado no código decompilado do DEX, através do índice denominado `fieldDexIndex`, como sendo da classe *`Ljava/lang/String`*. Esse endereço pode ser calculado adicionando o offset informado (0x14) ao endereço dos dados do objeto. Assim, para o primeiro objeto da lista, temos 0x1384d2c0+0x14 = 0x1384d2d4. Com esse endereço e sabendo o tipo do objeto, recuperamos os dados do objeto, conforme ilustrado na Figura 4.14.

```
>python vol.py --profile=LinuxLinuxGoldfish_3_4_67ARM -f
memdumpWhatsAppChat.lime -p 1206 -b 0x700c7000 art_extract_object_data -o
0x1384d2d4
Object Address: 0x1384d2d4L
Address for IMAGE BOOT: 0x700c7000L
(classtype not root)
00 12 8a 13 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 50 ae 9f 7f 53 01 00 00 00 00 00 00   ....P...S.......
00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

Figura 4.14 - Execução da ferramenta `art_extract_object_data` **para recuperação de dados de uma determinada propriedade do objeto analisado**

Nessa listagem, como se trata de um ponteiro, não é identificada uma classe para o objeto e a ferramenta apresenta um *dump* da região da memória próxima a partir do endereço fornecido. Identificado como um ponteiro, usamos o parâmetro "-c" da ferramenta, normalmente utilizado com o nome da classe para forçar uma interpretação específica dos dados, que nesse caso é utilizado o termo "pointer" que força a ferramenta a retornar o endereço do ponteiro:

```
>python vol.py --profile=LinuxLinuxGoldfish_3_4_67ARM -f
memdumpWhatsAppChat.lime -p 1206 -b 0x700c7000 art_extract_object_data -o
0x1384D2D4 -c pointer
Object Address: 0x1384d2d4L
Address for IMAGE BOOT: 0x700c7000L
Pointer 0x138a1200L
```

Figura 4.15 - Execução da ferramenta `art_extract_object_data` **para recuperação de dados de uma determinada propriedade do objeto analisado com decodificação por referência indireta**

Com o endereço do campo interno “I” da lista de campos recuperado a partir das meta-informações da classe do objeto, é recuperado os dados do campo invocando novamente a mesma ferramenta passando o endereço alvo 0x138a1200 como parâmetro, conforme ilustrado na Figura 4.16.

```
>python vol.py --profile=LinuxLinuxGoldfish_3_4_67ARM -f
memdumpWhatsAppChat.lime -p 1206 -b 0x700c7000 art_extract_object_data -o
0x138a1200
Object Address: 0x138a1200L
Address for IMAGE BOOT: 0x700c7000L
(classtype is root)
target_art_class 0x700df8f0L
Class name: Ljava/lang/String;
offset 0x138a1220L
count 0x9L
MESSAGE 1
```

Figura 4.16 - Execução da ferramenta `art_extract_object_data` **para recuperação de uma propriedade de texto do objeto**

Repetindo o processo para o campo denominado “K”, vigésimo primeiro campo interno da classe, pode recuperar o horário da mensagem através do endereço que pode ser calculado adicionando o offset informado (0x58) ao endereço dos dados do objeto. Assim, para mesma mensagem, temos 0x1384d2c0+0x58 = 0x1384d318. Podemos repetir a execução da ferramenta para esse endereço, conforme ilustrado na Figura 4.17.

```
>python vol.py --profile=LinuxLinuxGoldfish_3_4_67ARM -f
memdumpWhatsAppChat.lime -p 1206 -b 0x700c7000 art_extract_object_data -o
0x1384d318
Object Address: 0x1384d318L
Address for IMAGE BOOT: 0x700c7000L
(classtype not root)
50 ae 9f 7f 53 01 00 00 00 00 00 00 00 00 00 00   P...S...........
04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 0d 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00 b8 7c 8b 70 00 00 00 00   .........|.p....
50 0b 76 13 00 00 00 00 20 ef 7e 13 00 ef 7e 13   P.v.....~...~.
```

Figura 4.17 - Execução da ferramenta `art_extract_object_data` **para recuperação de uma propriedade de datação do objeto**

Como se trata de um tipo básico “J” (long), nesse endereço está armazenado diretamente o próprio valor, e sendo uma arquitetura ARM-32 (*little-endian*), o valor da datação da mensagem, presente nos oito primeiros bytes, é 0x01537f9fae50 (1458135084624), que corresponde a uma data em linux *epochtime*, que convertida para fuso horário do aparelho (confirmado pela propriedade recuperada `persist.sys.timezone`

como GMT), corresponde a 16 Mar 2016 13:31:24.624 GMT, exatamente como mostrado na tela do dispositivo.

A relação entre os objetos abordados no processo de recuperação de dados está ilustrada na Figura 4.18. Com os demais objetos de conversa recuperados dessa classe alvo e suas meta-informações sobre os campos, puderam ser identificados os demais campos referentes ao corpo da mensagem, datação, identificação do interlocutor das conversas do aplicativo (conforme Figura 4.19 contendo cópia da tela do aparelho com bate-papo ativo), e outras, confirmando a eficácia da técnica na recuperação de dados de objetos Java de uma determinada classe a partir de dados localizados na memória RAM.

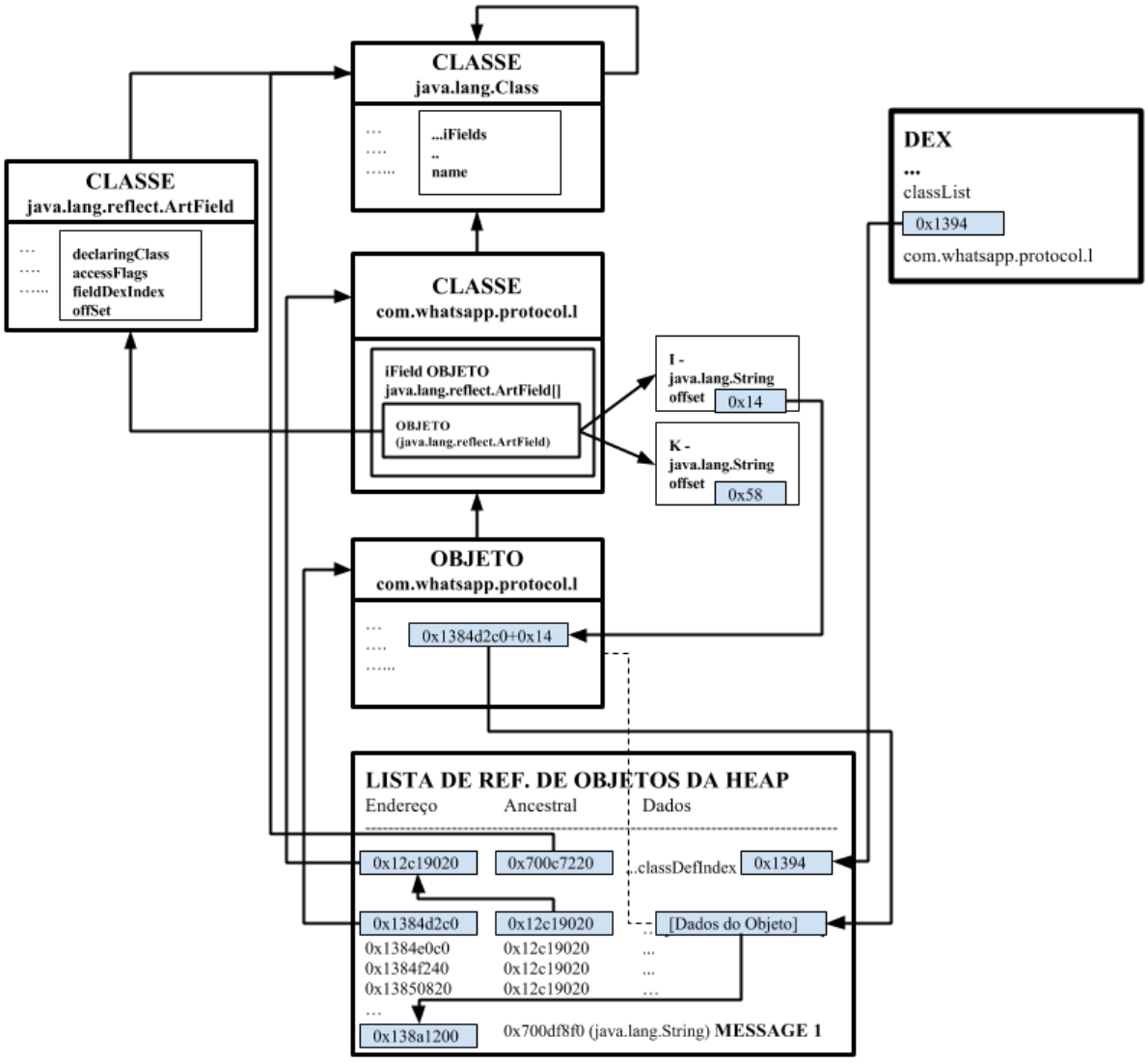


Figura 4.18- Relação entre os objetos utilizados no processo de recuperação

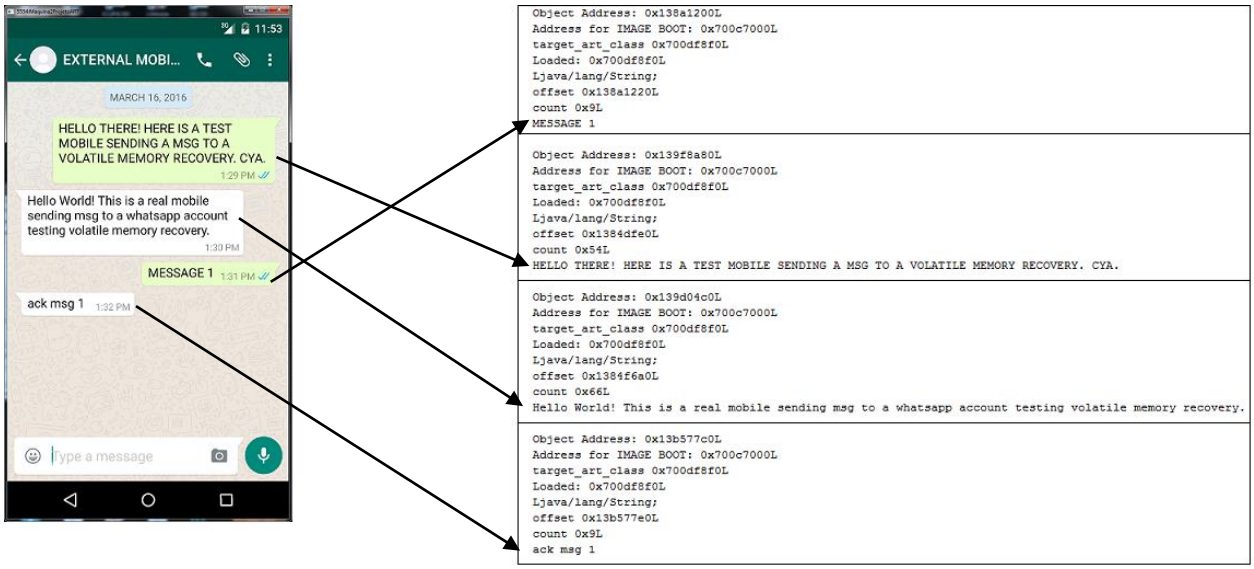


Figura 4.19 - Ilustração da recuperação dos textos da conversa do aplicativo de bate-papo

Para validação da técnica, além da análise detalhada descrita, também foi realizada a recuperação com sucesso de dados de objetos de interesse forense como contatos, mantidos na *heap* da aplicação *com.android.contacts*.

4.3. VALIDAÇÃO COM O DISPOSITIVO REAL

O dispositivo real se tratava de um Samsung Galaxy S4 (GT-I9500 non-LTE) com CPU Exynos 5410, 2 GB RAM, Android 5.0.1 (API level 21) original, *build number* LRX22C.I9500UBUHOL1, e vm.heapSize 64 MB.

4.3.1. CONFIGURAÇÃO DO AMBIENTE

O código fonte do kernel (versão 3.4.5) foi obtido do repositório site do fabricante (http://opensource.samsung.com). O aparelho continha diversas aplicações em execução, inclusive o aplicativo de bate-papo.

4.3.2. APLICAÇÃO DA TÉCNICA

Inicialmente, o processo para localização do processo alvo (*com.whatsapp*) foi executado e foram recuperados os dados a respeito do ambiente de execução da aplicação, como o endereço para o mapeamento do *framework* Android.

Em seguida, interessante notar que o cabeçalho da imagem do *framework* no dispositivo era diferente da do dispositivo emulado, apesar de constar no identificador do cabeçalho o mesmo número de versão (009).

Conforme se observa na Figura 4.20, examinando os dados do cabeçalho ART do aplicativo alvo no dispositivo real, o campo referente ao endereço da imagem não apontava para um endereço absoluto válido dentro do segmento de endereçamento da aplicação (0x025C3000). Essa diferença sugere que o sistema Android desse fabricante não corresponde ao código do AOSP.

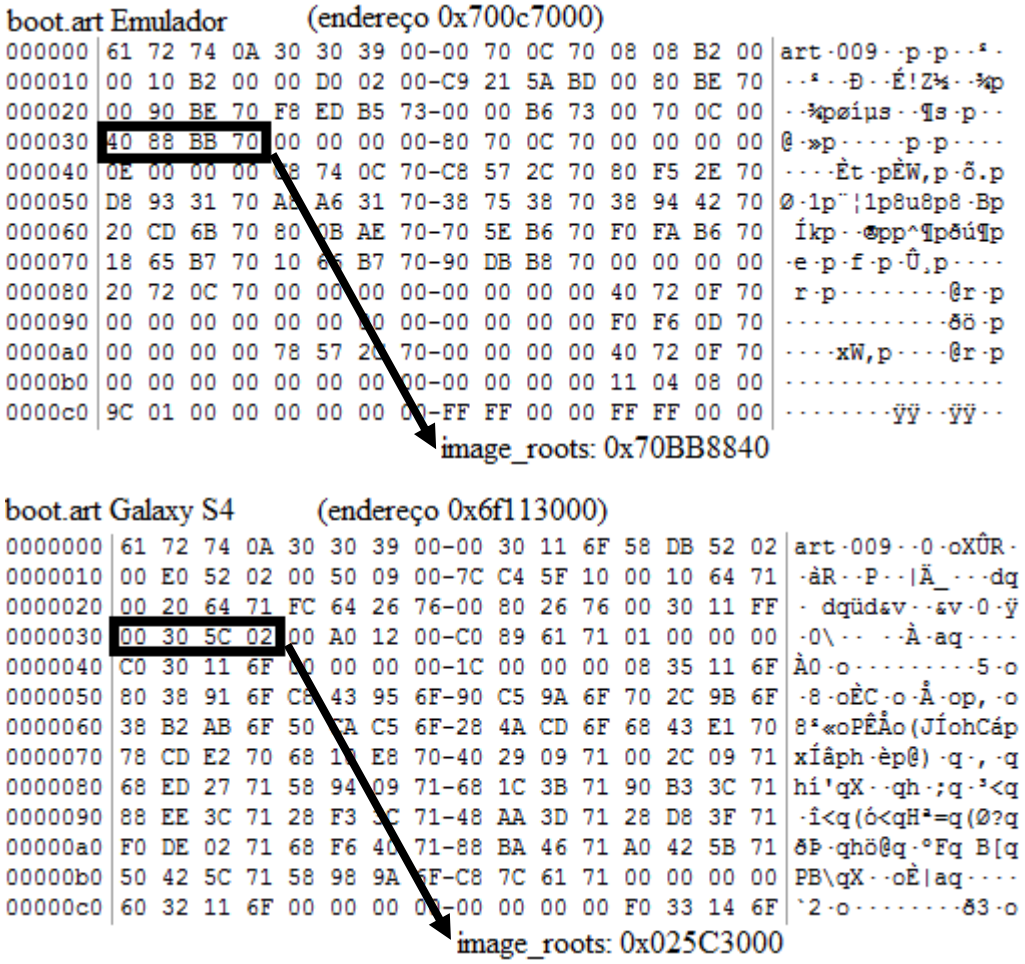


Figura 4.20 - Cabeçalhos das imagens ART dos dispositivos emulado e real (Samsung Galaxy S4).

Com isso, a técnica proposta não pode ser totalmente aplicada para o dispositivo examinado, pois a diferença detectada no cabeçalho utilizado pelo fabricante exige que seja efetuado um processo de engenharia reversa do ART em uso no dispositivo real. Este resultado de avaliação mostra uma limitação comum que caracteriza procedimentos projetados para a extração de objetos de sistemas operacionais em constante evolução como os dos dispositivos móveis. Além disso, a consequente exigência relativa à adaptação da

técnica proposta a esta nova situação confronta-se com um obstáculo importante, uma vez que não existe um código fonte público do ART fornecido pelo fabricante.

5. CONCLUSÃO

Este trabalho apresentou uma técnica para análise de dados de objetos presentes em aquisições de memória RAM de dispositivos operando o sistema Android na arquitetura ARM 32-bits. O trabalho inclui estudo de conceitos e estruturas do ambiente de execução ART presente no Android 5.0 disponível no AOSP. Foi realizada avaliação experimental da técnica proposta utilizando extensões construídas para uso com o *framework* Volatility.

A técnica proposta traz a contribuição para a comunidade forense na recuperação de potenciais evidências digitais para uma investigação com a possibilidade de se extrair e analisar objetos Java no ART revelando estruturas de objetos além do proposto em [HILGERS et al. 2014] (limitado para versões do Android inferiores a 5.0) e de outras técnicas tradicionais baseadas na detecção de padrões intrínsecos aos artefatos, como o realizado em [APOSTOLOPOULOS et al. 2013]. Uma contribuição adicional é a intenção de tornar as ferramentas desenvolvidas públicas e a possibilidade de seu uso para procedimentos de engenharia reversa de aplicações.

Importante notar que a técnica proposta e as ferramentas utilizadas foram construídas visando a flexibilidade em serem adaptadas para outras arquiteturas (incluindo 64-bits), para dispositivos com diferentes limitações de hardware, e para modificações já identificadas em versões mais modernas do Android (6.0). Ressalta-se que, apesar do sucesso da técnica efetuada sobre dispositivo emulado, foi detectado que a implementação do ART em um dispositivo real é diferente da versão AOSP testada no dispositivo emulado. Porém, o conhecimento da arquitetura do Android aprofundado nesse trabalho e a técnica desenvolvida servem como base para o aperfeiçoamento da computação forense no entendimento do Android em dispositivos reais e em futuras versões desse sistema operacional.

Como trabalhos futuros, se pretende realizar validações experimentais com outras marcas e modelos de dispositivos reais e utilizar a técnica no desenvolvimento de procedimentos de detecção e análise de malwares.

6. RESULTADOS ACADÊMICOS

Algumas dessas contribuições foram aceitas para publicação em congressos científicos, como o caso do artigo Análise de Objetos a partir da Extração da Memória RAM de Sistemas Sobre Android Runtime (ART), publicado e condecorado como *Best*-Paper na conferência IADIS Ibero-Americana WWW/Internet 2016 (CIAWI 2016) e do artigo *A Technique for Extraction and Analysis of Application Heap Objects within Android Runtime (ART)*, aceito para publicação no *3rd International Conference on Information Systems Security and Privacy* (ICISSP 2017).

7. REFERÊNCIAS BIBLIOGRÁFICAS

ANDROID OPEN SOURCE PROJECT - AOSP. <http://source.android.com/> Acesso em: 06 de junho de 2014.

APOSTOLOPOULOS, D. et al. (2013). Discovering authentication credentials in volatile memory of android mobile devices. Collaborative, Trusted and Privacy-Aware e/m-Services. Springer Berlin Heidelberg. p. 178-185.

ARON, L.; HANÁCEK, P. (2015). Introduction to Android 5 Security. SOFSEM (Student Research Forum Papers/Posters), p. 103-111.

BACKES, M. et al. (2016). ARTist: The Android Runtime Instrumentation and Security Toolkit. Cornell University Library. arXiv:1607.06619, 2016.

BECHTSOUDIS, A. (2015). "Fuzzing Objects d'ART - Digging Into the New Android L Runtime Internals". Whitepaper. Disponível em <http://census-labs.com/media/Fuzzing_Objects_d_ART_hitbsecconf2015ams_WP.pdf>. Acesso em: abril de 2015.

BREZINSKI, D.; KILLALEA, T. (202). Guidelines for evidence collection and archiving (RFC 3227).

CARRIER, B. (2003). Defining digital forensic examination and analysis tools using abstraction layers. International Journal of digital evidence, v. 1, n. 4, p. 1-12.

CASTRO, J. D. (2016) Introducing Linux Distros. Apress.

DRAKE, J. J. et al. (2014). Android hacker's handbook. John Wiley & Sons.

ELENKOV, N. (2014) Android Security Internals: An In-Depth Guide to Android's Security Architecture. No Starch Press.

GOOGLE. Android Developer. <https://developer.android.com/>. Acesso em: dezembro de 2016.

GOOGLE. The ART runtime. Vídeo apresentado por Brian Carlstrom, Anwar Ghuloum, Ian Rogers. Disponível em <https://www.google.com/events/io/io14videos/b750c8da-aebe-e311-b297-00155d5066d7>, 2014. Acesso em: abril 2015.

HILGERS, C. et al. (2014). Post-mortem memory analysis of cold-booted android devices. *IT Security Incident Management & IT Forensics* (IMF), *Eighth International Conference on*. IEEE, p. 62-75.

HØGSET, E. S. (2015). Investigating the security issues surrounding usage of Ephemeral data within Android environments. Master thesis. *UiT The Arctic University of Norway*.

KINGO. Disponível em <http://www.kingoapp.com/index.htm>. Acesso em: dezembro 2016.

LAUREANO, M. A. P.; MAZIERO, C. A. (2008). Virtualização: Conceitos e aplicações em segurança. Livro-Texto de Minicursos SBSeg, p. 1-50.

LIGH, M. H. et al. (2014). The art of memory forensics: detecting malware and threats in Windows, Linux, and Mac memory. John Wiley & Sons.

MCHOES, A.; FLYNN, I. M. (2011). Understanding operating systems. 6ª edição. Cengage Learning.

Open Handset Alliance. Disponível em: <http://www.openhandsetalliance.com>. Acesso em: maio de 2016.

PETSAS, T. et al. (2014). Rage against the virtual machine: hindering dynamic analysis of android malware. *Proceedings of the Seventh European Workshop on System Security*. ACM, p. 5.

SABANAL, P. (2014). State Of The ART. Exploring The New Android KitKat Runtime.

<https://conference.hitb.org/hitbsecconf2014ams/materials/D1T2-State-of-the-Art-Exploring-the-New-Android-KitKat-Runtime.pdf>. *5th Hack In The Box Conference. Haxpo 2014 Amsterdam*. Acesso em: setembro de 2015.

SABANAL, P. (2015). Hiding Behind ART, Paul Sabanal. <https://www.blackhat.com/docs/asia-15/materials/asia-15-Sabanal-Hiding-Behind-ART-wp.pdf>. *BlackHat 2015 Asia*. Acesso em: setembro de 2015.

SIMÃO, A. M. L.; SICOLI, F. C.; MELO, L. P.; SOUSA JR.; R. T. (2011). Acquisition of digital evidence in android smartphones. *Proceedings of the 9th Australian Digital Forensics Conference*, Edith Cowan University.

SILBERSCHATZ, A. et al. (2008). Sistemas operacionais com Java. 7ª Edição. Elsevier Brasil.

SYLVE, J. et al. (2012). Acquisition and analysis of volatile memory from android devices. Digital Investigation, v. 8, n. 3, p. 175-184.

VIDAS, T.; CHRISTIN, N. (2014). Evading android runtime analysis via sandbox detection. *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, p. 447-458.

VOLATILESYSTEMS. Volatility Project Homepage. <https://github.com/volatilityfoundation/volatility>. Acesso em: abril de 2015.

WÄCHTER, P.; GRUHN, M. (2015). Practicability study of android volatile memory forensic research. *Information Forensics and Security (WIFS), 2015 IEEE International Workshop on*. IEEE. p. 1-6.

YAGHMOUR, K. (2013). Embedded Android: Porting, Extending, and Customizing. O'Reilly Media, Inc.

APÊNDICES

A - COMPILAÇÃO DO MÓDULO KERNEL LIME

Para compilação do módulo de *kernel* LiME, segue-se os seguintes passos:

1. Obtenção do código fonte do módulo *kernel* que pode ser acessado por:

```
$ git clone https://github.com/504ensicsLabs/LiME.git
```

2. Inicialmente, preparar e instalar dependências para o ambiente de compilação conforme as instruções contidas em http://source.android.com/source/initializing.html

3. Download e descompressão do Android NDK contido em http://developer.android.com/tools/sdk/ndk/index.html

4. Download e descompressão do Android SDK contido em http://developer.android.com/sdk/index.html

5. Download e descompressão do código fonte do *kernel* para o dispositivo alvo, geralmente acessível em caminhos indicados pelo site do fabricante.

6. Rooteamento do aparelho. Conforme mencionado, o processo de rooteamento é intrinsicamente ligado às características do aparelho e essa etapa é necessária para execução do módulo *kernel*. Existem ferramentas online que analisam as configurações do dispositivo e efetuam processo de root explorando as vulnerabilidades detectadas. Uma ferramenta desse tipo está disponível para uso livre em http://kingoroot.org.

7. Preparação do ambiente de compilação:

   Para facilitar o processo de compilação, se pode definir as seguintes variáveis de ambiente:

```
export SDK_PATH=/path/to/android-sdk-linux/
export NDK_PATH=/path/to/android-ndk/
export KSRC_PATH=/path/to/kernel-source/
```

```
export  CC_PATH=$NDK_PATH/toolchains/arm-linux-androideabi-
4.4.3/prebuilt/linux-x86/bin/
export LIME_SRC=/path/to/lime/src
```

8. Preparação do código fonte do *kernel*.

   Recuperar o arquivo de configuração do *kernel* do sistema do dispositivo:

```
cd $SDK_PATH/platform-tools
./adb pull /proc/config.gz
gunzip ./config.gz
cp config $KSRC_PATH/.config
```

   Em seguida, preparação do código fonte do *kernel* para o módulo

```
$ cd $KSRC_PATH
$     make     ARCH=arm     CROSS_COMPILE=$CC_PATH/arm-eabi-
modules_prepare
```

9. Preparar a compilação do módulo.

Ajustar o arquivo `Makefile` para compilação cruzada do módulo. Um arquivo modelo é fornecido junto com o código fonte do LiME. O conteúdo do arquivo deve ser semelhante ao seguinte:

```
obj-m := lime.o
lime-objs := main.o tcp.o disk.o
KDIR := /path/to/kernel-source
PWD := $(shell pwd)
CCPATH := /path/to/android-ndk/toolchains/arm-linux-androideabi-
4.6/prebuilt/linux-x86/bin/
default:
 $(MAKE)  ARCH=arm  CROSS_COMPILE=$(CCPATH)/arm-eabi-  -C  $(KDIR)
M=$(PWD) modules
```

Depois, compilar o módulo:

```
cd $LIME_SRC
make
```

A.1 COMPILAÇÃO DO KERNEL

Em alguns casos, o *kernel* em operação no aparelho não foi compilado com os parâmetros que permitem a carga de módulos de *kernel*. Uma forma de verificar se o *kernel* permite carga de módulos é através do comando `ls /proc/modules`. Caso não permita, não será listado o recurso module. Nessa situação, uma forma de contorno é compilar um novo *kernel* para o dispositivo com os parâmetros de configuração a seguir no arquivo de configuração de compilação (defconfig):

```
CONFIG_MODULES=y
CONFIG_MODULES_UNLOAD=y
CONFIG_MODULES_FORCE_LOAD=y
CONFIG_MODULES_FORCE_UNLOAD=y
```

Para isso, o código fonte do *kernel* (incluindo drivers) do dispositivo alvo é necessário no processo.

Para validação experimental desse trabalho, foi necessário a substituição dos *kernels* originais dos dispositivos utilizados. Para o dispositivo emulado foi criada uma compilação doo código fonte do *kernel* para o emulador na versão 3.4.67, disponível no sítio do projeto AOSP, para o dispositivo Android ARM emulado (apelido *goldfish*) integrante das ferramentas no kit de desenvolvimento Android (diretório `[androidsdkhome]/sdk/tools`); e para o dispositivo real, foi criado um *kernel* utilizando o código fonte do kernel (versão 3.4.5) obtido do repositório site do fabricante (http://opensource.samsung.com).

O ambiente de compilação cruzada pode ser montado com os seguintes passos:

1. Inicialmente, preparar e instalar dependências para a ambiente compilação conforme as instruções contidas em http://source.android.com/source/initializing.html
2. Download e descompressão do Android NDK contido em http://developer.android.com/tools/sdk/ndk/index.html
3. Download e descompressão do Android SDK contido em http://developer.android.com/sdk/index.html

Uma descrição para compilação de diversos *kernel*s para dispositivos Android existe no site do projeto AOSP[18].

A.1.1 COMPILAÇÃO CRUZADA PARA AMBIENTE EMULADO

Com ambiente preparado pronto, a compilação do *kernel* pode ser obtida através da sequência de comandos:

```
# Obtenção do código fonte do Kernel
# No repositório AOSP kernel AOSP
#
$ git clone https://android.googlesource.com/kernel/goldfish.git
$ cd goldfish
$ git branch –a
* master
 remotes/origin/HEAD -> origin/master
 remotes/origin/android-goldfish-2.6.29
 remotes/origin/android-goldfish-3.4
 remotes/origin/linux-goldfish-3.0-wip
 remotes/origin/master
$ git checkout -t origin/android-goldfish-3.4 -b goldfish

# Instalação de dependência necessária em alguns ambientes
sudo apt-get install libncurses-dev

export SDK_PATH=/usr/share/android-sdk/
export NDK_PATH=/usr/share/android-ndk/
export KSRC_PATH=~/goldfish/
export            CC_PATH=$NDK_PATH/toolchains/arm-linux-androideabi-
4.6/prebuilt/linux-x86_64/bin/
export  EXTRA_CFLAGS='-fno-pic  -march=armv7-a  -mfloat-abi=softfp  -
mfpu=vfpv3-d16'

# Pegar o Kernel e compilar com config de kernel do goldfish
# existe um def config /goldfish/arch/arm/configs
# goldfish_armv7_defconfig
```

[18] https://source.android.com/source/building-kernels.html

```
export ARCH=arm
export SUBARCH=arm
export CROSS_COMPILE=$CC_PATH/arm-linux-androideabi-

# no defconfig
# colocar opção de LKM (Load Modules Kernels)
# pode ser via menuconfig
make menuconfig
# colocar as seguintes propriedades
#  CONFIG_MODULES=y
#  CONFIG_MODULES_UNLOAD=y
#  CONFIG_MODULES_FORCE_LOAD=y
#  CONFIG_MODULES_FORCE_UNLOAD=y
# compilar
make goldfish_armv7_defconfig
```

Depois de compilado, executa-se o emulador configurado para o *kernel* compilado:

```
# Executar emulador com arquivo do kenel como parâmetro
$SDK_PATH\sdk\tools>emulator -avd AndroidRuntimeAVD -kernel zImage
```

Conectado no dispositivo emulado, via *Android Debug Bridge*[19] (ADB), podemos verificar a versão do *kernel* em execução:

```
root@generic:/ # cat /proc/version
Linux version 3.4.67-g7fda5cc-dirty (santoku@santoku-VirtualBox) (gcc
version 4.6 20120106 (prerelease) (GCC) ) #2 PREEMPT Thu Sep 17
11:13:44 BRT 2015
```

A 1.2. COMPILAÇÃO CRUZADA PARA DISPOSITIVO REAL

Para o dispositivo real é feito o procedimento de compilação cruzada semelhante para o emulado, utilizando o código fonte específico para o aparelho.

Para uso do *kernel*, é necessário que seja feito um processo de gravação na partição de *bootloader* do aparelho contendo imagem do *kernel* compilado. Para isso,

[19] Android Debug Bridge (ADB) é uma ferramenta de linha de commando que permite a comunicação via porta USB com um dispositivo Android com o serviço de depuração inicializado.

são utilizadas ferramentas específicas para cada marca e modelo de aparelho. Para o aparelho utilizado na avaliação da técnica, foi feita gravação utilizando o aplicativo de código aberto RASHR[20], com capacidade de efetuar o processo de gravação e cópia de segurança do *kernel* para um grande espectro de modelo de aparelhos Android que estejam "rooteados".

Após a gravação, e reinicialização do aparelho conectado no dispositivo real utilizado nesse trabalho, via ADB, podemos verificar, após substituição do *kernel*, a versão em execução:

```
shell@ja3g:/ $ cat /proc/version
Linux version 3.4.5-g5bc8393-dirty (santoku@santoku-VirtualBox) (gcc
version 4.6 20120106 (prerelease) (GCC) #6 SMP PREEMPT Wed Oct 12
00:11:40 BRT 2015
```

A.2 UTILIZAÇÃO DO MÓDULO LIME

O módulo LiME pode receber alguns parâmetros que definem destino dos dados adquiridos. Os dados podem ser transmitidos por TCP ou gravados em um dispositivo de armazenamento secundário, comumente um cartão de memória, tipo *microSD*, inserido no dispositivo.

Para extração através do TCP, pode ser realizada a seguinte sequência de comandos que em essência enviam o módulo compilado para o armazenamento secundário do dispositivo e mapeiam o socket TCP do dispositivo para uma porta específica da estação conectada:

```
adb push lime.ko /sdcard/lime.ko
adb forward tcp:4444 tcp:4444
adb shell
su
#
```

No dispositivo, a instalação no módulo *kernel* pode ser feita através dos seguintes comandos:

[20] Código fonte disponível em https://github.com/dslnexus/Rashr

```
insmod /sdcard/lime.ko "path=tcp:4444 format=lime"
```

Após a instalação, o módulo fica preparado para iniciar a extração quando for conectado na porta configurada. Na estação que receberá os dados, se executa o seguinte comando que inicia a extração no dispositivo:

```
nc localhost 4444 > ram.lime
```

Visando uma preservação dos buffers de rede ou outras informações que podem ser perdidas com uso dos módulos de rede, também é possível realizar a extração destinando os dados para o dispositivo de armazenamento secundário através do parâmetro `path` informado no momento da instalação do módulo:

```
insmod /sdcard/lime.ko "path=/sdcard/ram.lime format=lime"
```

Nesse caso, no momento da instalação é iniciado o processo de extração.

Com o acesso aos dados de memória RAM extraídos, esses podem ser analisados com ferramentas específicas. Para esse trabalho, os dados foram extraídos e armazenados no formato LiME, que insere cabeçalhos com informações sobre o espaço de endereçamento dos segmentos de memória e pode ser analisado pelas extensões disponíveis para o *framework* Volatility. Mas, além desse formato de extração, o módulo também permite a extração dos dados brutos (formato *raw*) ou com *padding* de bytes 0 (formato *padded*) inserido nos segmentos de endereços mapeados para dispositivos ou outra finalidade diferente do armazenamento de memória RAM.

Para as aquisições procedidas nesse trabalho, foi realizada a transferência dos dados adquiridos via TCP visando preservar áreas de memória relacionadas à manutenção dos arquivos mapeados.

B – TABELAS

Tabela B.0.1 - Algumas extensões para plataforma Linux contidas no *framework* Volatility

```
>python ./vol.py --info |grep linux
Volatility Foundation Volatility Framework 2.4
linux_arp            - Imprime a tabela ARP
...
linux_dmesg          - Coleta o buffer do dmesg
linux_dump_map       - Grava mapeamentos de memória específicos no disco
linux_elfs           - Localiza binários ELF nos mapeamentos do processo
…
linux_find_file      - Lista e recupera arquivos da memoria
…
linux_ifconfig       - Coleta sobre as interfaces ativas
…
linux_iomem          - Saída similar ao /proc/iomem
…
linux_library_list   - Lista bibliotecas carregas em um processo
linux_librarydump    - Extrai bibliotecas compartilhadas da memória de um
processo para o disco
…
linux_proc_maps      - Coleta informações de mapeamentos de memória de um
processo
…
linux_procdump        - Extrai imagem do executável de um processo para o
disco
…
linux_psaux          - Coleta informação dos processos incluindo linha de
commando e hora de início de execução
 …
linux_pslist      - Coleta informações sobre as tareas ativas percorrendo
a task_struct->task list
..
linux_psxview    - Localiza processos escondidos com listas de processos
…
```

C – CÓDIGOS-FONTE

C.1 ART_VTYPES.PY

```
"""
@author:       Alberto Magno
@contact:      alberto.magno@gmail.com
"""

import volatility.obj as obj

art_vtypes = {
    # ####################################
    # runtime/oat.h
    #
    'ArtHeader' : [ 0x08, {
        'magic' : [ 0x0, ['array', 4, ['char']]],
        'version' : [ 0x4, ['array', 4, ['char']]],
        'image_begin' : [ 0x8, ['unsigned int']],
        'image_size' : [ 0x0c, ['unsigned int']],
        'image_bitmap_offset' : [ 0x10, ['unsigned int']],
        'image_bitmap_size' : [ 0x14, ['unsigned int']],
        'oat_checksum' : [ 0x18, ['unsigned int']],
        'oat_file_begin' : [ 0x1c, ['unsigned int']],
        'oat_data_begin' : [ 0x20, ['unsigned int']],
        'oat_data_end' : [ 0x24, ['unsigned int']],
        'oat_file_end' : [ 0x28, ['unsigned int']],
        'patch_delta' : [ 0x2c, ['int']],
        'image_roots' : [ 0x30, ['unsigned int']],
        'compile_pic' : [ 0x34, ['unsigned int']],
        }],
    # runtime/oat.h
    #
    'OatHeader' : [ 0x54, {
        'magic' : [ 0x0, ['array', 4, ['char']]],
        'version' : [ 0x4, ['array', 4, ['char']]],
        'adler32_checksum' : [ 0x8, ['unsigned int']],
        'instruction_set' : [ 0x0c, ['unsigned int']],
        'instruction_set_features' : [ 0x10, ['unsigned int']],
        'dex_file_count' : [ 0x14, ['unsigned int']],
        'executable_offset' : [ 0x18, ['unsigned int']],
        'interpreter_to_interpreter_bridge_offset' : [ 0x1c, ['unsigned int']],
        'interpreter_to_compiled_code_bridge_offset' : [ 0x20, ['unsigned int']],
        'jni_dlsym_lookup_offset_' : [ 0x24, ['unsigned int']],
        'portable_imt_conflict_trampoline_offset' : [ 0x28, ['unsigned int']],
        'portable_resolution_trampoline_offset' : [ 0x2c, ['unsigned int']],
        'portable_to_interpreter_bridge_offset' : [ 0x30, ['int']],
        'quick_generic_jni_trampoline_offset' : [ 0x34, ['unsigned int']],
        'quick_imt_conflict_trampoline_offset' : [ 0x38, ['unsigned int']],
        'quick_resolution_trampoline_offset' : [ 0x3c, ['unsigned int']],
        'quick_to_interpreter_bridge_offset' : [ 0x40, ['unsigned int']],
        'image_patch_delta' : [ 0x44, ['int']],
        'image_file_location_oat_checksum' : [ 0x48, ['unsigned int']],
        'image_file_location_oat_data_begin' : [ 0x4c, ['unsigned int']],
        'key_value_store_size' : [ 0x50, ['unsigned int']],
        'key_value_store' : [ 0x54, ['array', lambda x: x.key_value_store_size,
['char']]],
        }],

    # runtime/Object.h
    #
    'OatDexHeader' : [ 0x10, {
        'dex_file_location_size' : [ 0x0, ['unsigned int']],
        'dex_file_location_data' : [ 0x4, ['array', lambda x: x.dex_file_location_size,
['char']]],
        #'dex_file_location_checksum' : [ 0x8, ['unsigned int']],
        #'dex_file_pointer' : [ 0x0c, ['unsigned int']],
        #'classes_offsets' : [ 0x10, ['array', lambda x: x.dex_file_location_size,
```

```
['unsigned int']]],
        }],

    'DexHeader' : [ 0x70, {
        'dex_file_magic' : [ 0x0, ['array', 0x8, ['char']]],
        'checksum' : [0x8,['unsigned int']],
        'signature':[0xc,['array',0x14,['unsigned char']]],
        'file_size' : [0x20,['unsigned int']],
        'header_size' : [0x24,['unsigned int']],
        'endian_tag' : [0x28,['unsigned int']],
        'link_size' : [0x2c,['unsigned int']],
        'link_off' : [0x30,['unsigned int']],
        'map_off' : [0x34,['unsigned int']],
        'string_ids_size' : [0x38,['unsigned int']],
        'string_ids_off' : [0x3c,['unsigned int']],
        'type_ids_size' : [0x40,['unsigned int']],
        'type_ids_off' : [0x44,['unsigned int']],
        'proto_ids_size' : [0x48,['unsigned int']],
        'proto_ids_off' : [0x4c,['unsigned int']],
        'field_ids_size' : [0x50,['unsigned int']],
        'field_ids_off' : [0x54,['unsigned int']],
        'method_ids_size' : [0x58,['unsigned int']],
        'method_ids_off' : [0x5c,['unsigned int']],
        'class_defs_size' : [0x60,['unsigned int']],
        'class_defs_off' : [0x64,['unsigned int']],
        'data_size' : [0x68,['unsigned int']],
        'data_off' : [0x6c,['unsigned int']],
        }],

    'DexMapItem' : [ 0xc, {
        'type' : [0x0,['unsigned short']],
        'unused' : [0x2,['unsigned short']],
        'sizeof' : [0x4,['unsigned int']],
        'offset' : [0x8,['unsigned int']],
        }],

    'DexMap' : [ 0x4, {
        'count' : [0x0,['unsigned int']],
        'values' : [ 0x4, ['array', lambda x: x.count, ['DexMapItem']]],
        }],

    'OatClassHeader' : [ 0x08, {
        'status' : [ 0x0, ['short int']],
        'type' : [0x4,['short']],
        'bitmapsize':[0x8,['unsigned int']],
        'bitmap' : [0xc,['array', lambda x: x.bitmapsize, ['char']]],
        'methods_offsets' : [0x24,['array', lambda x: x.bitmapsize, ['char']]],
        }],

    '[Ljava/lang/Object;' : [ 0x10, {
        'shadow_klass_' : [ 0x0, ['unsigned int']],
        'shadow_monitor_' : [0x4,['unsigned int']],
        'length' : [ 0x8, ['unsigned int']],
        'values' : [0xc,['array', lambda x: x.length , ['unsigned int']]],
        }],

    '[Ljava/lang/reflect/ArtField;' : [ 0x10, {
        'shadow_klass_' : [ 0x0, ['unsigned int']],
        'shadow_monitor_' : [0x4,['unsigned int']],
        'length' : [ 0x8, ['unsigned int']],
        'values' : [0xc,['array', lambda x: x.length , ['pointer',
['Ljava/lang/reflect/ArtField;']]]],
        }],

        'Z' : [ 0x8, {
        'value1' : [ 0x0, ['unsigned int']],
        'value2' : [0x4,['unsigned int']],
        }],
  ######################################################################
    #runtime/mirror/class.h
    'Ljava/lang/Class;' : [ 0x64, {
        'classLoader':[0x0,['unsigned int']], #efining class loader, or NULL for the
"bootstrap" system loader
        'componentType' : [0x4,['unsigned int']], #For array classes, the component
```

```
class object for instanceof/checkcast (for String[][][], this will be String[][]). NULL
for non-array classes.
        'dexCache' : [0x8,['unsigned int']], #DexCache of resolved constant pool
entries (will be NULL for classes generated by the runtime such as arrays and primitive
classes).
        'directMethods' : [0xc,['unsigned int']], #static, private, and <init> methods
        'iFields' : [0x10,['unsigned int']], #instance fields These describe the layout
of the contents of an Object. Note that only the fields directly declared by this class
are listed in ifields; fields declared by a superclass are listed in the superclass's
Class.ifields. All instance fields that refer to objects are guaranteed to be at the
beginning of the field list.  num_reference_instance_fields_ specifies the number of
reference fields.
        'ifTable' : [0x14,['unsigned int']], #The interface table (iftable_) contains
pairs of a interface class and an array of the interface methods. There is one pair per
interface supported by this class.  That means one pair for each interface we support
directly, indirectly via superclass, or indirectly via a superinterface.  This will be
null if neither we nor our superclass implement any interfaces. Why we need this: given
"class Foo implements Face", declare "Face faceObj = new Foo()". Invoke faceObj.blah(),
where "blah" is part of the Face interface.  We can't easily use a single vtable. For
every interface a concrete class implements, we create an array of the concrete vtable_
methods for the methods in the interface.
        'imTable' : [0x18,['unsigned int']],# Interface method table (imt), for quick
"invoke-interface".
        'name' : [0x1c,['pointer', ['Ljava/lang/String;']]],#Descriptor for the class
such as "java.lang.Class" or "[C". Lazily initialized by ComputeName
        'sFields' : [0x20,['unsigned int']], #Static fields.
        'superClass' : [0x24,['unsigned int']], #The superclass, or NULL if this is
java.lang.Object, an interface or primitive type.
        'verifyErrorClass' : [0x28,['unsigned int']], #If class verify fails, we must
return same error on subsequent tries.
        'virtualMethods' : [0x2c,['unsigned int']], #Virtual methods defined in this
class; invoked through vtable.
        'vtable' : [0x30,['unsigned int']], # Virtual method table (vtable), for use by
"invoke-virtual".  The vtable from the superclass is copied in, and virtual methods
from our class either replace those from the super or are appended. For abstract
classes, methods may be created in the vtable that aren't in virtual_ methods_ for
miranda methods.
        'accessFlags' : [0x34,['unsigned int']], # Access flags; low 16 bits are
defined by VM spec.
        'classSize' : [0x38,['unsigned int']], # Total size of the Class instance; used
when allocating storage on gc heap. See also object size .
        'clinitThreadId' : [0x3c,['unsigned int']], # Tid used to check for recursive
<clinit> invocation.
        'dexClassDefIndex' : [0x40,['unsigned int']], # ClassDef index in dex file, -1
if no class definition such as an array.
        'dexTypeIndex' : [0x44,['unsigned int']], # Type index in dex file.
        'numReferenceInstanceFields' : [0x48,['unsigned int']], #Number of instance
fields that are object refs.
        'numReferenceStaticFields' : [0x4c,['unsigned int']], #Number of static fields
that are object refs,
        'objectSize' : [0x50,['unsigned int']],#Total object size; used when allocating
storage on gc heap.(For interfaces and abstract classes this will be zero.)See also
class size .
        'primitiveType' : [0x54,['unsigned int']], #Primitive type value, or
Primitive::kPrimNot (0); set for generated primitive classes.
        'referenceInstanceOffsets' : [0x58,['unsigned int']], #Bitmap of offsets of
ifields.
        'referenceStaticOffsets' : [0x5c,['unsigned int']], #Bitmap of offsets of
sfields.
        'status' : [0x60,['unsigned int']], #State of class
initialization.kStatusRetired = -2,
kStatusError = -1,kStatusNotReady = 0, kStatusIdx = 1(Loaded, DEX idx in
super class type idx  and interfaces type idx ). kStatusLoaded = 2(DEX idx values
resolved.) kStatusResolving = 3 (Just cloned from temporary class object.)
kStatusResolved = 4( Part of linking.), kStatusVerifying = 5( In the process of being
verified.),kStatusRetryVerificationAtRuntime = 6 (Compile time verification failed,
retry at runtime.kStatusVerifyingAtRuntime = 7 (Retrying verification at runtime.)
kStatusVerified = 8 (Logically part of linking; done pre-init.), kStatusInitializing =
9 (Class init in progress.),kStatusInitialized = 10(Ready to go.),kStatusMax = 11
        }],

    # runtime/mirror/art_field.h
    # mirror of java.lang.reflect.ArtField
    'Ljava/lang/reflect/ArtField;' : [ 0x10, {
```

```
        'declaringClass':[0x0,['pointer', ['Ljava/lang/Class;']]], #The class we are a
part of
        'accessFlags' : [0x4,['unsigned int']],
        'fieldDexIndex' : [0x8,['unsigned int']], #Dex cache index of field id
        'offset' : [0xc,['unsigned int']], #Offset of field within an instance or in
the Class' static fields
        }],
    # runtime/mirror/art method.h
    # mirror of java.lang.reflect.ArtMethod
    'Ljava/lang/reflect/ArtMethod;' : [ 0x40, {
        'declaringClass':[0x0,['pointer', ['Ljava/lang/Class;']]],#Field order required
by test "ValidateFieldOrderOfJavaCppUnionClasses".The class we are a part of.
        'dexCacheResolvedMethods' : [0x4,['unsigned int']],# Short cuts to
declaring class ->dex cache  member for fast compiled code access.
        'dexCacheResolvedTypes' : [0x8,['unsigned int']], #Short cuts to
declaring_class_->dex_cache_ member for fast compiled code access.
        'dexCacheStrings' : [0xc,['unsigned int']],# Short cuts to declaring_class_-
>dex cache  member for fast compiled code access.
        #64 bits
        'entryPointFromInterpreter' : [0x10,['long long']], # Method dispatch from the
interpreter invokes this pointer which may cause a bridge into compiled code.
        #64 bits
        'entryPointFromJni' : [0x18,['long long']],# Pointer to JNI function registered
to this method, or a function to resolve the JNI function.
        #64 bits
        #if defined(ART_USE_PORTABLE_COMPILER) uint64_t
entry_point_from_portable_compiled_code_;#endif
        'entryPointFromQuickCompiledCode' : [0x20,['long long']],# Method dispatch from
quick compiled code invokes this pointer which may cause bridging into portable
compiled code or the interpreter.
        #64 bits
        'gcMap' : [0x28,['long long']],#Pointer to a data structure created by the
compiler and used by the garbage collector to determine which registers hold live
references to objects within the heap. Keyed by native PC offsets for the quick
compiler and dex PCs for the portable.
        'accessFlags' : [0x2c,['unsigned int']],# Access flags; low 16 bits are defined
by spec.
        # Dex file fields. The defining dex file is available via declaring_class_-
>dex cache
        'dexCodeItemOffset' : [0x30,['unsigned int']],#Offset to the CodeItem.
        'dexMethodIndex' : [0x34,['unsigned int']],#Index into method_ids of the dex
file associated with this method.
        'methodIndex' : [0x38,['unsigned int']], # Entry within a dispatch table for
this method. For static/direct methods the index is into the
declaringClass.directMethods, for virtual methods the vtable and for interface methods
the ifTable.

        }],
    # runtime/mirror/array.h
    '[L' : [ 0x8, {
        'length' : [ 0x0, ['unsigned int']], #The number of array elements.
        'values' : [0x4,['array', lambda x: x.length , ['unsigned int']]], # Marker for
the data (used by generated code)
        }],

    '[C' : [ 0x8, {
        'length' : [ 0x0, ['unsigned int']], #The number of array elements.
        'array' : [0x4,['array', lambda x: x.length * 2 , ['char']]], # Marker for the
data (used by generated code)
        }],

    # runtime/mirror/dex cache.h
    # mirror of java.lang.DexCache.
    'Ljava/lang/DexCache;' : [ 0x1c, {
        'dex':[0x0,['unsigned int']],
        'location' : [0x4,['unsigned int']],
        'resolvedFields' : [0x8,['unsigned int']],
        'resolvedMethods' : [0xc,['unsigned int']],
        'resolvedTypes' : [0x10,['unsigned int']],
        'strings' : [0x14,['unsigned int']],
        # 64 bits
        'dexFile' : [0x18,['long long']],
        }],
```

```
    #runtime/mirror/object.h
    #mirror of java.lang.Object
    'Ljava/lang/Object;' : [ 0x8, {
        'klass' : [ 0x0, ['unsigned int']],
        'monitor' : [0x4,['unsigned int']],
        }],
        #ifdef USE BAKER OR BROOKS READ BARRIER Note names use a 'x' prefix and the
x_rb_ptr_ is of type int instead of Object to go with the alphabetical/by-type field
order on the Java side.
        #x_rb_ptr_ : [ 0x0, ['unsigned int']],      # For the Baker or Brooks pointer.
        #x xpadding_ : [ 0x0, ['unsigned int']],    # For 8-byte alignment. TODO: get
rid of this.
        #endif

    #runtime/mirror/proxy.h
    #mirror of java.lang.reflect.Proxy
    'Ljava/lang/reflect/Proxy;' : [ 0x4, {
        'h' : [ 0x0, ['unsigned int']],
        }],
    #runtime/mirror/reference.h
    #mirror of java.lang.ref.Reference
    'Ljava/lang/ref/Reference;' : [ 0xc, {
        'next' : [ 0x0, ['unsigned int']], #Note this is Java volatile:
        'prev' : [ 0x4, ['unsigned int']],#Note this is Java volatile:
        'zombie' : [ 0x8, ['unsigned int']],#Note this is Java volatile:
        }],

    # runtime/mirror/stack trace element.h
    # mirror of java.lang.StackTraceElement
    'Ljava/lang/StackTraceElement;' : [ 0x10, {
        'declaringClass':[0x0,['pointer', ['Ljava/lang/Class;']]], #['unsigned int']],
        'fileName' : [0x4,['pointer', ['Ljava/lang/String;']]],#['unsigned int']],
        'methodName' : [0x8,['pointer', ['Ljava/lang/String;']]],#['unsigned int']],
        'lineNumber' : [0xc,['unsigned int']],
        }],

    # runtime/mirror/throwable.h
    # mirror of java.lang.Throwable
    'Ljava/lang/Throwable;' : [ 0x10, {
        'cause':[0x0,['pointer', ['Ljava/lang/Class;']]], #['unsigned int']],
        'detail_message' : [0x4,['pointer', ['Ljava/lang/String;']]],#['unsigned
int']],
        'stack_trace' : [0x8,['pointer', ['Ljava/lang/String;']]],#['unsigned int']],
        'suppressed_exceptions' : [0xc,['unsigned int']],
        }],

    # runtime/mirror/string.h
    # mirror of java.lang.String
    'Ljava/lang/String;' : [ 0x08, {
        'offset' : [0x0,['unsigned int']],
        'count' : [0x4,['unsigned int']],
        'array' : [0x8,['array', lambda x: x.count * 2, ['char']]],
       # 'value' : [0x0,['unsigned int']],
       # 'count' : [0x4,['unsigned int']],
       # 'hashCode' : [0x8,['unsigned int']],
       # 'offset' : [0xc,['unsigned int']],
       # 'base klass' : [ 0x10, ['unsigned int']],
       # 'base klass monitor' : [0x14,['unsigned int']],
       # 'length' : [0x18,['unsigned int']],
       # 'array' : [0x1c,['array', lambda x: x.count * 2, ['char']]],
        }],
    # runtime/gc/allocator/rosalloc.h
    'rosalloc/runfixedhead' : [ 0x8, {
        'magic_num_' : [0x0,['unsigned char']],
        'size_bracket_idx_' : [0x1,['unsigned char']],
        'is_thread_local_' : [0x2,['unsigned char']],
        'to_be_bulk_freed_' : [0x3,['unsigned char']],
        'first_search_vec_idx_' : [ 0x4, ['unsigned int']],
        #'alloc_bit_map_' : [0x14,['unsigned int']],
        }],

    # 8K pagemaprun
    'rosalloc/runpagemap' : [ 0x2000, { ## reading just 8K
```

```
        'map' : [0x0,['array', 0x2000, ['unsigned char']]],
        }],

    # /libc/bionic/system_properties.cpp
    'prop_area' : [ 0x80, {
        'bytes_used' : [0x0,['unsigned int']],
        'serial' : [0x4,['unsigned int']],
        'magic' : [0x8,['unsigned int']],
        'version' : [0xc,['unsigned int']],
        'reserved' : [0x7c, ['unsigned int']],
        }],
    # /libc/bionic/system_properties.cpp
    'prop_bt' : [ 0x14, {
        'namelen' : [0x0,['unsigned char']],
        'reserved' : [0x1,['unsigned char']],
        'prop' : [0x4,['unsigned int']],
        'left' : [0x8,['unsigned int']],
        'right' : [0xc, ['unsigned int']],
        'children' : [0x10, ['unsigned int']],
        'name' : [0x14,['array',  lambda x: x.namelen, ['char']]],
        }],
    # /libc/bionic/system_properties.cpp
    'prop_info' : [ 0x60, {
        #'serial' : [0x0,['unsigned int']],
        'value_len' : [0x3,['unsigned char']],
        'value' : [0x4,['unsigned int']], # libc/include/sys/system_properties.h/
#PROP_VALUE_MAX = 96
        'name' : [0x60,['unsigned int']],
        }],
  }

class OatDexHeader(obj.CType):

    def classes_offsets(self, oatSize):
        class_count = 0
        offset = 0x0
        base = 4+self.dex_file_location_size+4+4
#sizeof(dex_file_location_size)+sizeof(dex_file_location_data)+sizeof(dex_file_location
 checksum)+size0f(dex file pointer)
        while offset < self.dex_file_pointer()-(base+4)-oatSize:
            class_offset = obj.Object('unsigned int', offset = self.v()+base+offset, vm
= self.obj_vm)
            offset += 0x4
            class_count += 1
            yield class_offset
    def dex_file_location_checksum(self):
        return obj.Object('unsigned int', offset =
self.v()+4+self.dex_file_location_size, vm = self.obj_vm)
    def dex_file_pointer(self):
        return obj.Object('unsigned int', offset =
self.v()+4+self.dex_file_location_size+4, vm = self.obj_vm)



class DalvikObjectClasses(obj.ProfileModification):
    conditions = {'os': lambda x: x == 'linux'}
    before = ['LinuxObjectClasses']

    def modification(self, profile):
        profile.vtypes.update(art_vtypes)
        profile.object_classes.update({'OatDexHeader': OatDexHeader})
```

C.2 ART_PROPERTIES_DATA.PY

```
"""
@author:       Alberto Magno
@contact:      alberto.magno@gmail.com
"""
import volatility.obj as obj
import volatility.plugins.linux.flags as linux_flags
import volatility.plugins.linux.common as linux_common
import volatility.plugins.linux.proc_maps as linux_proc_maps

class art_extract_properties_data(linux_common.AbstractLinuxCommand):

    def __init__(self, config, *args, **kwargs):
        linux_common.AbstractLinuxCommand.__init__(self, config, *args, **kwargs)
        config.add_option('PID', short_option = 'p', default = None,
                          help = 'Operate on this Process ID',
                          action = 'store', type = 'str')

    # Method for walk-through trie structures collecting leaf (prop_info) data
    def read_prop_bt(self, vm, address, properties):
        base = self.base
        prop_bt = obj.Object('prop_bt', vm = vm, offset = address)
        if (prop_bt.left > 0):
            self.read_prop_bt(vm, prop_bt.left+base, properties)
        if (prop_bt.children > 0):
            self.read_prop_bt(vm, prop_bt.children+base, properties)
        if (prop_bt.right > 0):
            self.read_prop_bt(vm, prop_bt.right+base, properties)
        if (prop_bt.prop>0):
            prop_info = obj.Object('prop_info', vm = vm, offset = prop_bt.prop+base)
            propertyName = obj.Object('String', vm = vm, offset = prop_bt.prop+base+96,
length = 32)
            propertyValue = obj.Object('String', vm = vm, offset = prop_bt.prop+base+4,
length = int(prop_info.value_len))
            properties.append((propertyName,propertyValue))


    def calculate(self):
        offset = 0x0
        mytask = None
        proc_maps = linux_proc_maps.linux_proc_maps(self._config).calculate()
        # Search for properties shared map in process maps
        for task, vma in proc_maps:
            if not vma.vm_file:
                continue
            if not (vma.vm_flags & 0x00000001 and not vma.vm_flags & 0x00000002 and not
vma.vm_flags & 0x00000004):
                continue
            if not linux_common.get_path(task, vma.vm_file) == "/dev/__properties__":
                continue
            mytask = task
            break
        proc_as = mytask.get_process_address_space()
        stringObject = None
        offset = vma.vm_start
        retorno=[]
        prop_area = obj.Object('prop_area', vm = proc_as, offset = offset)
        offset = offset+int(prop_area.size())
        self.base = offset
        properties=[]
        self.read_prop_bt(proc_as,self.base, properties)
        return properties

    def render_text(self, outfd, data):
        self.table_header(outfd, [("[Property]", "35"),("[value]", "82")])
        for property in data:
            self.table_row(outfd, "["+property[0]+"]=", "["+property[1]+"]")
```

C.3 ART_FIND_OAT.PY

```
"""
@author:       Alberto Magno
@contact:      alberto.magno@gmail.com
"""

import volatility.obj as obj
import volatility.plugins.linux.flags as linux_flags
import volatility.plugins.linux.common as linux_common
import volatility.plugins.linux.proc_maps as linux_proc_maps
import sys, traceback

class art_find_oat(linux_common.AbstractLinuxCommand):
    def __init__(self, config, *args, **kwargs):
        linux_common.AbstractLinuxCommand.__init__(self, config, *args, **kwargs)
        config.add_option('PID', short_option = 'p', default = None, help = 'Operate on
this Process ID', action = 'store', type = 'str')

    def calculate(self):
        mytasks = []
        proc_maps = linux_proc_maps.linux_proc_maps(self._config).calculate()
        for task, vma in proc_maps:
            if not vma.vm_file:
                continue
            memmap_file_path = linux_common.get_path(task, vma.vm_file)
            print memmap_file_path
            if not memmap_file_path.endswith(".dex"):
                continue
            if not (vma.vm_flags & 0x00000001 and not vma.vm_flags & 0x00000002 and not
vma.vm_flags & 0x00000004):
                continue
            mytasks.append([memmap_file_path,task,vma])


        retorno=[]
        for path,mytask,vma in mytasks:
            proc_as = mytask.get_process_address_space()
            print path, hex(vma.vm_start)
            oatHdr = obj.Object('OatHeader', vm = proc_as, offset =
vma.vm_start+0x1000)
            print oatHdr.magic
            if oatHdr.magic==['o','a','t','\x0A']:
                retorno.append((path,vma.vm_start))
        return retorno
    def render_text(self, outfd, data):
        self.table_header(outfd, [("oat", "128"),
                                  ("offset_", "32")])

        for map in data:
            self.table_row(outfd, map[0],
                                  hex(map[1]))
```

C.4 ART_DUMP_ROSALLOC_HEAP.PY

```
"""
@author:       Alberto Magno
@contact:      alberto.magno@gmail.com
"""

import volatility.obj as obj
import volatility.plugins.linux.flags as linux_flags
import volatility.plugins.linux.common as linux_common
import volatility.plugins.linux.proc_maps as linux_proc_maps
import volatility.plugins.linux.art_extract_object_data as art_extract_object_data

import volatility.utils as utils
import sys, traceback, string, struct



class art_dump_rosalloc_heap_objects(linux_common.AbstractLinuxCommand):
    # Root classes
    class_root_table = {};
    # Root classes's identifiers
    class_roots_descriptors = ["Ljava/lang/Class;",
                                    "Ljava/lang/Object;",
                                    "[Ljava/lang/Class;",
                                    "[Ljava/lang/Object;",
                                    "Ljava/lang/String;",
                                    "Ljava/lang/DexCache;",
                                    "Ljava/lang/ref/Reference;",
                                    "Ljava/lang/reflect/ArtField;",
                                    "Ljava/lang/reflect/ArtMethod;",
                                    "Ljava/lang/reflect/Proxy;",
                                    "[Ljava/lang/String;",
                                    "[Ljava/lang/reflect/ArtField;",
                                    "[Ljava/lang/reflect/ArtMethod;",
                                    "Ljava/lang/ClassLoader;",
                                    "Ljava/lang/Throwable;",
                                    "Ljava/lang/ClassNotFoundException;",
                                    "Ljava/lang/StackTraceElement;",
                                    "Z",
                                    "B",
                                    "C",
                                    "D",
                                    "F",
                                    "I",
                                    "J",
                                    "S",
                                    "V",
                                    "[Z",
                                    "[B",
                                    "[C",
                                    "[D",
                                    "[F",
                                    "[I",
                                    "[J",
                                    "[S",
                                    "[Ljava/lang/StackTraceElement;"
                                    ];
    kBitsPerByte = 8
    # art\runtime\globals.h
    kPageSize = 4096
    KB = 1024
    # art\runtime\thread.h kNumRosAllocThreadLocalSizeBrackets = 34 =
kNumOfSizeBrackets
    kNumOfSizeBrackets = 34
    pageMapSize = 0x2000000
```

```
    #TODO:test to LOCATE AES-256 key (experimental)
    key = '\x61\x00\x63\x00\x6b\x00\x20\x00\x6d\x00\x73\x00\x67\x00\x20\x00\x31'

    # Load de parent class
    def loadObjectShadowKlass(self,objAddress, vm):
        #print "objAddress",hex(objAddress)
        art_object = obj.Object('Ljava/lang/Object;', vm = vm, offset = objAddress)
        #print "carregando shadow klass desse objecto:", hex(art_object.klass)
        if art_object.klass <= 0x0 or not int(art_object.klass) in
self.class_root_table.keys():
            return None,None
        object_klass_name = self.class_root_table[int(art_object.klass)]
        return art_object,obj.Object(object_klass_name, vm = vm, offset = objAddress+8)

    # Load the class name based on parent class
    def loadObjectName(self,objAddress, vm):
        if objAddress<=0x0:
            return None
        address = int(obj.Object('unsigned int', vm = vm, offset = objAddress))
        while not address in art_extract_object_data.class_root_table.keys():
            print "classtype not root:", hex(address), " classlinker error - loading
parent class ",
            #TODO: search DEX classes :??
            #load parent class.
            address = obj.Object('unsigned int', vm = vm, offset = address)
        return art_extract_object_data.class_root_table[int(address)]

    # Round functions for adjusts in byte padding
    def roundDown(self, x, n):
        return (x & -n)
    def roundUp(self, x, n):
        return self.roundDown(x + n - 1, n)

    def __init__(self, config, *args, **kwargs):
        linux_common.AbstractLinuxCommand.__init__(self, config, *args, **kwargs)

        config.add_option('PID', short_option = 'p', default = None, help = 'Operate on
this Process ID', action = 'store', type = 'str')

        config.add_option('DEX_CLASSDEF_IDX', short_option = 'c', help = 'This is the
index (in hex) of the class in dex file to search', action = 'store', type = 'long',
metavar="HEX")
        config.add_option('SLOTBYTE_SEARCH', short_option = 't', help = 'This is the
index (in hex) of the class type in dex file to search', action = 'store', type =
'long', metavar="HEX")

        config.add_option('CLASS_ADDRESS_FILTER', short_option = 'o', help = 'This is
the  address (virtual address in hex) of class objects to search', action = 'store',
type = 'long', metavar="HEX")

        config.add_option('BOOT_ART_ADDRESS', short_option = 'b', help = 'This is the
base address (virtual address in hex) where image object data can be found', action =
'store', type = 'long', metavar="HEX")

        config.add_option('ROSALLOC_HEAP_ADDRESS', short_option = 'e', default =
"0x12c00000", help = 'This is the address (virtual address in hex) where the rosalloc
space heap data can be found',  action = 'store', type = 'long', metavar="HEX")

        config.add_option('ROSALLOC_PAGEMAP_ADDRESS', short_option = 'm', default =
"0xb1d70000", help = 'This is the address (virtual address in hex) where the rossaloc
pagemap data can be found', action = 'store', type = 'long', metavar="HEX")

        config.add_option('ROSALLOC_PAGEMAP_SIZE', short_option = 's', default =
0x2000000, help = 'This is the size (in hex) of the rossaloc pagemap data', action =
'store', type = 'long', metavar="HEX")

    def calculate(self):


        if not self._config.PID:
            print "This plugin requires a PID to be given via the '-p' switch option"
```

```
            return
        proc_maps = linux_proc_maps.linux_proc_maps(self._config).calculate()

        for task, vma in proc_maps:
            if str(task.pid) == str(self._config.PID):
                break
            continue

        #Loading root classes references
        #*********************************************************************
        bootArt = obj.Object('ArtHeader', vm = task.get_process_address_space(), offset
= self._config.BOOT_ART_ADDRESS)
        imageRoots = obj.Object('[Ljava/lang/Object;', vm =
task.get_process_address_space(), offset = int(bootArt.image_roots))
        kClassRoots = obj.Object('[Ljava/lang/Object;', vm =
task.get_process_address_space(), offset = int(imageRoots.values[7]))
        class_count = 0;
        for class_descriptor in self.class_roots_descriptors:
            object_klass_name = kClassRoots.values[class_count]
            class_count = class_count + 1
            self.class_root_table[int(object_klass_name)]=class_descriptor

        rosalloc_heap_address =  self._config.ROSALLOC_HEAP_ADDRESS
#int(self._config.ROSALLOC_HEAP_ADDRESS,16)

        # Loading heap page map
        #print "Loading heap page map"
        pageMap = obj.Object('Array', count = self.pageMapSize, targetType = "unsigned
char", vm = task.get_process_address_space(), offset =
self._config.ROSALLOC_PAGEMAP_ADDRESS)
        '''
        kPageMapReleased = 0,     // Zero and released back to the OS.
        kPageMapEmpty = 1,            // Zero but probably dirty.
        kPageMapRun = 2,              // The beginning of a run.
        kPageMapRunPart = 3,          // The non-beginning part of a run.
        kPageMapLargeObject = 4,      // The beginning of a large object.
        kPageMapLargeObjectPart = 5,  // The non-beginning part of a large object.
        '''
        allocRuns = []
        runCount = 0

        returnData = []
        for pageOffset in
range(0x0,self._config.ROSALLOC_PAGEMAP_SIZE,self.kPageSize):#self.pageMapSize):
            #print '#Page address:',hex(self._config.ROSALLOC_HEAP_ADDRESS+pageOffset),
            pageIdx = pageOffset/self.kPageSize
            runfixedheader = obj.Object('rosalloc/runfixedhead', vm =
task.get_process_address_space(), offset = rosalloc_heap_address+pageOffset)
            numOfPages = 0;
            bracket_idx = runfixedheader.size_bracket_idx_
            # numOfPages based in bracket index.
            if (bracket_idx < 4):
              numOfPages = 1
            else:
                if (bracket_idx < 8):
                    numOfPages = 2
                else:
                    if (bracket_idx < 16):
                        numOfPages = 4
                    else:
                        if (bracket_idx < 32):
                            numOfPages = 8
                        else:
                            if (bracket_idx == 32):
                                numOfPages = 16
                            else:
                                numOfPages = 32

            # read run all data
            #currentRun = bytearray()
            proc_as = task.get_process_address_space()
            currentRun = proc_as.zread(rosalloc_heap_address+pageOffset,
```

```
numOfPages*self.kPageSize)

            # bracketSize.
            if (bracket_idx < self.kNumOfSizeBrackets - 2):
                bracket_size = 16 * (bracket_idx + 1)
            else:
                if (bracket_idx == self.kNumOfSizeBrackets - 2):
                    bracket_size = 1 * self.KB
                else:
                    bracket_size = 2 * self.KB

            '''kPageMapReleased = 0,     // Zero and released back to the OS.
                kPageMapEmpty = 1,            // Zero but probably dirty.
                kPageMapRun = 2,              // The beginning of a run.
                kPageMapRunPart = 3,          // The non-beginning part of a run.
                kPageMapLargeObject = 4,      // The beginning of a large object.
                kPageMapLargeObjectPart = 5,  // The non-beginning part of a large
object.'''

            '''if pageMap[pageIdx]==0:
                print ' (kPageMapReleased)'
            else:
                if pageMap[pageIdx]==1:
                    print ' (kPageMapEmpty)'
                else:
                    if pageMap[pageIdx]==2:
                        print ' (kPageMapRun)'
                    else:
                        if pageMap[pageIdx]==3:
                            print ' (kPageMapRunPart)'
                        else:
                            if pageMap[pageIdx]==4:
                                print ' (kPageMapLargeObject)'
                            else:
                                if pageMap[pageIdx]==5:
                                    print ' (kPageMapLargeObjectPart)'''

            if pageMap[pageIdx] == 2: #TODO: or pageMap[pageIdx] == 4: #for LOS
                #print "pageMap[pageIdx]=2"
                run_size = numOfPages*self.kPageSize
                allocRuns.append(run_size)
                runCount = runCount + 1

                if runfixedheader.size_bracket_idx_ == None:
                    print ''#     null data page'
                else:
                    max_slots = run_size / bracket_size
                    for tmp_slots_count in range(max_slots,0,-1):
                        tmp_slots_size = tmp_slots_count * bracket_size
                        bitmap_length =
self.roundUp(tmp_slots_count,self.kBitsPerByte*4)/self.kBitsPerByte
                        # Find the right number of slots, that is, there was enough
space for the header (including the bit maps.)
                        tmp_unaligned_header_size = self.kBitsPerByte+3*bitmap_length

                        tmp_header_size = 0
                        # Align up the unaligned header size. bracket_size may not be a
power of two.
                        if tmp_unaligned_header_size % bracket_size == 0:
                            tmp_header_size = tmp_unaligned_header_size
                        else:
                            tmp_header_size = tmp_unaligned_header_size + (bracket_size
- tmp_unaligned_header_size % bracket_size)
                        #print
'tmp header size:',tmp header size,'tmp slots count:',tmp slots count
                        if tmp_slots_count*bracket_size + tmp_header_size <= run_size:
                            header_size = tmp_header_size + run_size % bracket_size
                            #print '     MAGIC NUM:',runfixedheader.magic_num_
                            #print '     BRACKET IDX:',runfixedheader.size_bracket_idx_
                            #print '     IS THREAD
LOCAL:',runfixedheader.is thread local
                            #print '     TO BE BULK
```

```
FREED:',runfixedheader.to_be_bulk_freed_
                            #print '     FIRST SEARCH VEC
IDX:',runfixedheader.first search vec idx
                            #print '     Total num pages:', numOfPages
                            #print '     Bracket size:', bracket_size
                            #print '     size:',run_size

                            #if runfixedheader.first_search_vec_idx_>0:
                            #    header size =
self.kBitsPerByte*2*(runfixedheader.first_search_vec_idx_)

                            #print '     header_size:',header_size,
                            #print '     slots:', tmp_slots_count

                            #print '     SLOTS:'

                            for slot_idx in range(0,tmp_slots_count,1):
                                #print "     slot",slot_idx, hex(
rosalloc_heap_address+pageOffset+header_size+bracket_size*slot_idx),
                                #print '(page
offset',hex(header size+bracket size*slot idx),')'
                                currentSlot =
bytes(currentRun[header_size+bracket_size*slot_idx:header_size+bracket_size*slot_idx+br
acket_size])
                                #dump slot
                                if self.isNotEmpty(currentSlot):

                                    klass_shadow =
struct.unpack("i",currentSlot[0:4])[0]
                                    baseObjectAddress =
rosalloc_heap_address+pageOffset+header_size+bracket_size*slot_idx
                                    if self._config.SLOTBYTE_SEARCH:
                                        found = currentSlot.find(self.key) #reverse[::-
1])
                                        if found>0:
                                            returnData.append((baseObjectAddress,
pageIdx, bracket_idx, slot_idx, "KEY FOUND ADDRESS at "+found+" byte",
"****************************************"))
                                    currentSlotText = ""
                                    for offset, hexchars, chars in
utils.Hexdump(currentSlot):
                                        currentSlotText = currentSlotText + "{1:<48}
{2}".format(header_size+bracket_size*slot_idx+offset, hexchars, ''.join(chars))
                                    #print baseObjectAddress, pageIdx, bracket_idx,
slot_idx, hex(klass_shadow),currentSlotText

                                    if int(klass_shadow) in
self.class_root_table.keys(): #is a known root class, put classs name
                                        if
self.class_root_table[int(klass_shadow)]=='Ljava/lang/String;':
                                            stringObj =
obj.Object('Ljava/lang/String;', vm = proc_as, offset = baseObjectAddress+8)
                                            #print hex(stringObj.offset)
                                            #print hex(stringObj.count)
                                            charArrayObj = obj.Object('[C', vm =
proc_as, offset = stringObj.offset+8)
                                            stringValue = obj.Object('String', vm =
proc_as, offset = stringObj.offset+8+4,length=charArrayObj.length*2,encoding="utf16")
                                            #stringValue = obj.Object('String', vm
= proc as, offset = baseObjectAddress+8+0x8,length=stringObj.count*2,encoding="utf16")
                                            #print stringValue
                                            returnData.append((baseObjectAddress,
pageIdx, bracket_idx, slot_idx,
self.class_root_table[int(klass_shadow)]+"("+hex(klass_shadow)+")", stringValue))
                                            continue
                                        if
self.class_root_table[int(klass_shadow)]=='[C':
                                            stringObj = obj.Object('[C', vm =
proc_as, offset = baseObjectAddress+8)
                                            stringValue = obj.Object('String', vm =
proc_as, offset = baseObjectAddress+8+4,length=stringObj.length*2,encoding="utf16")
                                            returnData.append((baseObjectAddress,
```

```
pageIdx, bracket_idx, slot_idx,
self.class_root_table[int(klass_shadow)]+"("+hex(klass_shadow)+")", stringValue))
                                                    continue
                                                if
self.class_root_table[int(klass_shadow)]=='Ljava/lang/Class;':
                                                    classObj =
obj.Object('Ljava/lang/Class;', vm = proc_as, offset = baseObjectAddress+8)
                                                    if not classObj.name == 0x0:
                                                        stringObj =
obj.Object('Ljava/lang/String;', vm = proc_as, offset = classObj.name)
                                                        charArrayObj = obj.Object('[C', vm
= proc_as, offset = stringObj.offset+8)
                                                        stringValue = obj.Object('String',
vm = proc_as, offset =
stringObj.offset+8+4,length=charArrayObj.length*2,encoding="utf16")
                                                    else:
                                                        StringObj = "*No name loaded*"
                                                    print hex(baseObjectAddress), pageIdx,
bracket_idx, slot_idx,
self.class_root_table[int(klass_shadow)]+"("+hex(klass_shadow)+")", "
dexCache:"+hex(classObj.dexCache)+" name:"+stringValue+"
dexClassDefIndex:"+hex(classObj.dexClassDefIndex)+" iFields:"+hex(classObj.iFields)
                                                    returnData.append((baseObjectAddress,
pageIdx, bracket_idx, slot_idx,
self.class_root_table[int(klass_shadow)]+"("+hex(klass_shadow)+")", "
dexCache:"+hex(classObj.dexCache)+" name:"+stringValue+"
dexClassDefIndex:"+hex(classObj.dexClassDefIndex)+" iFields:"+hex(classObj.iFields)))
                                                    continue
                                                # dump slot data
                                                returnData.append((baseObjectAddress, pageIdx,
bracket_idx, slot_idx,
self.class_root_table[int(klass_shadow)]+"("+hex(klass_shadow)+")", currentSlotText))
                                                continue
                                            else: #not a root object class
                                                # recovery parent data class
                                                object_klass, upper_klass =
self.loadObjectShadowKlass(klass_shadow, task.get_process_address_space())

                                                if not object_klass==None and
int(object_klass.klass) in self.class_root_table.keys():
                                                #upper_klass.dexClassDefIndex == 0x1394 and
upper klass.dexTypeIndex == 0x1810:
                                                    if
self.class_root_table[int(object_klass.klass)]=='Ljava/lang/Class;':# and
upper_klass.dexClassDefIndex == self._config.DEX_CLASSDEF_IDX: #and
upper_klass.dexTypeIndex == self._config.DEX_TYPE_IDX: #parent class
                                                        classObj =
obj.Object('Ljava/lang/Class;', vm = proc_as, offset = object_klass.klass)
                                                        stringObj =
obj.Object('Ljava/lang/String;', vm = proc_as, offset = classObj.name)
                                                        charArrayObj = obj.Object('[C', vm =
proc_as, offset = stringObj.offset+8)
                                                        stringValue = obj.Object('String', vm =
proc_as, offset = stringObj.offset+8+4,length=charArrayObj.length*2,encoding="utf16")
                                                        returnData.append((baseObjectAddress,
pageIdx, bracket_idx, slot_idx, "*unknow*(parent
java.lang.class)"+"("+hex(klass_shadow)+")", "Parent
dexCache:"+hex(classObj.dexCache)+" name:"+stringValue+"
dexClassDefIndex:"+hex(classObj.dexClassDefIndex)+" iFields:"+hex(classObj.iFields)))
                                                        #returnData.append((baseObjectAddress,
pageIdx, bracket_idx, slot_idx, "*(FOUND)* "+hex(klass_shadow),currentSlotText))
                                                    else:
                                                        returnData.append((baseObjectAddress,
pageIdx, bracket_idx, slot_idx, "*unknow
class*"+"("+hex(klass_shadow)+")",currentSlotText))
                                                else:
                                                    #print '********** Empty slot **********'
                                                    pass
                                                #break #volatile
                                            break
        return returnData
    def isNotEmpty(self, data):
```

```
        notEmpty = 0
        for x in data:
            if ord(x) == 0:
                continue
            else:
                notEmpty = 1
                break
        return notEmpty

    def render_text(self, outfd, data):
        self.table_header(outfd, [("address", "14"), ("page", "7"), ("bracket", "4"),
("slot", "4"), ("obj class", "32"),("data", "120")])
        for columm in data:
            self.table_row(outfd, hex(columm[0]), columm[1], columm[2], columm[3],
columm[4], columm[5])
```

C.5 ART_EXTRACT_OBJECT_DATA.PY

```
"""
@author:       Alberto Magno
@contact:      alberto.magno@gmail.com
"""

import volatility.obj as obj
import volatility.plugins.linux.flags as linux_flags
import volatility.plugins.linux.common as linux_common
import volatility.plugins.linux.proc_maps as linux_proc_maps
import volatility.scan as scan
import volatility.utils as utils
import sys, traceback, string
import volatility.plugins.linux.art_dex_class_data as dexHandler


class art_extract_object_data(linux_common.AbstractLinuxCommand):
    class_root_table = {};
    class_roots_descriptors = ["Ljava/lang/Class;",
                                    "Ljava/lang/Object;",
                                    "[Ljava/lang/Class;",
                                    "[Ljava/lang/Object;",
                                    "Ljava/lang/String;",
                                    "Ljava/lang/DexCache;",
                                    "Ljava/lang/ref/Reference;",
                                    "Ljava/lang/reflect/ArtField;",
                                    "Ljava/lang/reflect/ArtMethod;",
                                    "Ljava/lang/reflect/Proxy;",
                                    "[Ljava/lang/String;",
                                    "[Ljava/lang/reflect/ArtField;",
                                    "[Ljava/lang/reflect/ArtMethod;",
                                    "Ljava/lang/ClassLoader;",
                                    "Ljava/lang/Throwable;",
                                    "Ljava/lang/ClassNotFoundException;",
                                    "Ljava/lang/StackTraceElement;",
                                    "Z",
                                    "B",
                                    "C",
                                    "D",
                                    "F",
                                    "I",
                                    "J",
                                    "S",
                                    "V",
                                    "[Z",
                                    "[B",
                                    "[C",
                                    "[D",
                                    "[F",
                                    "[I",
                                    "[J",
                                    "[S",
                                    "[Ljava/lang/StackTraceElement;"
                                    ];

    def __init__(self, config, *args, **kwargs):
        linux_common.AbstractLinuxCommand.__init__(self, config, *args, **kwargs)
        config.add_option('PID', short_option = 'p', default = None, help = 'Operate on
this Process ID', action = 'store', type = 'str')
        config.add_option('OBJECT_ADDRESS', short_option = 'o',
                          help = 'This is the address (in hex) where the object data can be
found',
                          action = 'store', type = 'long', metavar="HEX")
        config.add_option('BOOT_ART_ADDRESS', short_option = 'b',
                          help = 'This is the address (in hex) where the object data can be
found',
                          action = 'store', type = 'long', metavar="HEX")
```

```
        config.add_option('DEX_ADDRESS', short_option = 'x',
                          help = 'This is the address (in hex) where the mapped dex file
data can be found',
                          action = 'store', type = 'long', metavar="HEX")
        config.add_option('CLASS_NAME', short_option = 'c',
                          help = 'This is the name (registred in art_vtypes) of the object
data layout described',
                          action = 'store', type = 'str')

    def loadObjectShadowKlass(self,objAddress, vm):
        art_object = obj.Object('Ljava/lang/Object;', vm = vm, offset = objAddress)
        if art_object.klass <= 0x0:
            return None
        object_klass_name =
art_extract_object_data.class_root_table[int(art_object.klass)]
        return obj.Object(object_klass_name, vm = vm, offset = objAddress+8)

    def loadObjectName(self,objAddress, vm):
        if objAddress<=0x0:
            return None
        address = int(obj.Object('unsigned int', vm = vm, offset = objAddress))
        while not address in art_extract_object_data.class_root_table.keys():
            print "classtype not root:", hex(address), " classlinker error - loading
parent class ",
            #TODO: search DEX classes :??
            #load parent class.
            address = obj.Object('unsigned int', vm = vm, offset = address)

        return art_extract_object_data.class_root_table[int(address)]

    def retriveObjectDataPointer(self, obj_address):
        print 'Pointer address:',hex(obj_address)
        object = obj.Object('Ljava/lang/Object;', vm = self.vm, offset = obj_address)
        print 'object_klass :',hex(object.klass)
        if object == 0:
            print '          null'
            return
        return object.klass

    def retriveObjectData(self, obj_address, object_klass_name):

        print "Class name:",object_klass_name
        # Verify if object is an array
        if len(object_klass_name)> 1 and object_klass_name[:2] == '[L':
            count = 1
            #print  'Values:',
            art_object = obj.Object('[L', vm = self.vm, offset = obj_address+8)
            print hex(obj_address+8)
            object_array = []
            length = art_object.length
            for object in art_object.values:
                object_array.append(object)

            print  'Values:','['+str(art_object.length)+']'

            if object_klass_name=="[Ljava/lang/String;":
                length = art_object.length
                for object in object_array:
                    print count,'->',
                    string_object =
art_extract_object_data.loadObjectShadowKlass(self,int(object), self.vm)
                    if string_object is None:
                        print hex(object), "memory data not present"
                    else:
                        self.retriveObjectData(object,"Ljava/lang/String;")
                        #print hex(object), filter(lambda x: x in string.printable,
''.join(str(c) for c in string_object.array))
                    count = count + 1

            elif object_klass_name=="[Ljava/lang/reflect/ArtField;":
                length = art_object.length
                print length
                #TODO: Load all DEX? - Loading specific DEX - provisory
                #*****************************************************************
```

```
                # recover DEX file address in app vm
                if not self._config.DEX_ADDRESS==None:
                    self.dex = dexHandler.art_dex_class_data(self._config.DEX_ADDRESS,
int(self._config.PID))
                for object in object_array:
                    print count,'->',hex(object),
                    artfield_object =
art_extract_object_data.loadObjectShadowKlass(self,int(object), self.vm)
                    if not artfield_object == None:
                        print ' declaringClass',hex(artfield_object.declaringClass)
                        print ' accessFlags', hex(artfield_object.accessFlags)

                        if not self._config.DEX_ADDRESS==None:
                            field_desc =
self.dex.getDexField(artfield_object.fieldDexIndex)
                            print ' fieldDexIndex', hex(artfield_object.fieldDexIndex),
field_desc
                        else:
                            print ' fieldDexIndex', hex(artfield_object.fieldDexIndex)

                        field_vm_address = obj_address+artfield_object.offset-4
                        print ' offset', hex(artfield_object.offset), "recover field
data at (${base object address}+"+hex(artfield_object.offset)+" )"
                    else:
                        print "empty field data"
                    count = count + 1

            elif object_klass_name=="[Ljava/lang/reflect/ArtMethod;":
                #print object klass name
                length = art_object.length
                for object in object_array:
                    print count,'->',
                    artmethod_object =
art_extract_object_data.loadObjectShadowKlass(self,int(object), self.vm)
                    if artmethod_object is None:
                        print hex(int(object)), "memory data not present"
                    else:
                        if artmethod_object.declaringClass is None:
                            print hex(object),'-
>',artmethod_object.declaringClass.dexCache, ''.join(str(c) for c in
artmethod_object.declaringClass.name.array)
                        else:
                            print hex(object),'->','declaringClass',"memory data not
present"
                    count = count + 1
            else:
                for object in object_array:
                    print count,'-
>',hex(object),art_extract_object_data.loadObjectName(self,int(object),self.vm)
                    count = count + 1

        else: #not array objects
            art_object = obj.Object(object_klass_name, vm = self.vm, offset =
obj_address+8)

            if object_klass_name=='Ljava/lang/Class;':
                print 'classLoader',hex(art_object.classLoader)
                print 'componentType' ,hex(art_object.componentType)
                print 'dexCache' ,hex(art_object.dexCache),
art_extract_object_data.loadObjectName(self,int(art_object.dexCache),self.vm)
                print 'directMethods' ,hex(art_object.directMethods),
art_extract_object_data.loadObjectName(self,int(art_object.directMethods),self.vm)
                print 'iFields'
,hex(art_object.iFields),art_extract_object_data.loadObjectName(self,int(art_object.iFi
elds),self.vm)
                print 'ifTable'
,hex(art_object.ifTable),art_extract_object_data.loadObjectName(self,int(art_object.ifT
able),self.vm)
                print 'imTable'
,hex(art_object.imTable),art_extract_object_data.loadObjectName(self,int(art_object.imT
able),self.vm)
                print 'name'
,hex(art_object.name),art_extract_object_data.loadObjectName(self,int(art_object.name),
self.vm),''.join(str(c) for c in art_object.name.array)
```

```
                print 'sFields'
,hex(art_object.sFields),art_extract_object_data.loadObjectName(self,int(art_object.sFi
elds),self.vm)
                print 'superClass'
,hex(art_object.superClass),art_extract_object_data.loadObjectName(self,int(art_object.
superClass),self.vm)
                print 'verifyErrorClass'
,hex(art_object.verifyErrorClass),art_extract_object_data.loadObjectName(self,int(art_o
bject.verifyErrorClass),self.vm)
                print 'virtualMethods'
,hex(art_object.virtualMethods),art_extract_object_data.loadObjectName(self,int(art_obj
ect.virtualMethods),self.vm)
                print 'vtable'
,hex(art_object.vtable),art_extract_object_data.loadObjectName(self,int(art_object.vtab
le),self.vm)
                print 'accessFlags' ,hex(art_object.accessFlags)
                print 'classSize' ,hex(art_object.classSize)
                print 'clinitThreadId' ,hex(art_object.clinitThreadId)
                print 'dexClassDefIndex' ,hex(art_object.dexClassDefIndex)
                print 'dexTypeIndex' ,hex(art_object.dexTypeIndex)
                print 'numReferenceInstanceFields'
,hex(art_object.numReferenceInstanceFields)
                print 'numReferenceStaticFields'
,hex(art_object.numReferenceStaticFields)
                print 'objectSize' ,hex(art_object.objectSize)
                print 'primitiveType' ,hex(art_object.primitiveType)
                print 'referenceInstanceOffsets'
,hex(art_object.referenceInstanceOffsets)
                print 'referenceStaticOffsets' ,hex(art_object.referenceStaticOffsets)
                print 'status' ,hex(art_object.status)

            if object_klass_name=="Ljava/lang/String;":
                print "offset",hex(art_object.offset)
                print "count",hex(art_object.count)
                charArrayObj = obj.Object('[C', vm = self.vm, offset =
art_object.offset+8)
                stringValue = obj.Object('String', vm = self.vm, offset =
art_object.offset+8+4,length=art_object.count*2,encoding="utf16")
                print stringValue

            if object_klass_name=="[C":
                stringObj = obj.Object('[C', vm = self.vm, offset = obj_address+8)
                stringValue = obj.Object('String', vm = self.vm, offset =
obj_address+8+4,length=stringObj.length*2,encoding="utf16")
                print "count",hex(stringObj.length)
                print stringValue

            if object_klass_name=="Ljava/lang/DexCache;":
                print 'dex', hex(art_object.dex)
                #print 'location', hex(art_object.location)
                stringObj = obj.Object('Ljava/lang/String;', vm = self.vm, offset =
art_object.location+8)
                charArrayObj = obj.Object('[C', vm = self.vm, offset =
stringObj.offset+8)
                stringValue = obj.Object('String', vm = self.vm, offset =
stringObj.offset+8+4,length=stringObj.count*2,encoding="utf16")

                print 'location', stringValue+"(Ljava/lang/String; at
"+hex(art_object.location)+")"
                print 'resolvedFields', hex(art_object.resolvedFields)
                print 'resolvedMethods', hex(art_object.resolvedMethods)
                print 'resolvedTypes', hex(art_object.resolvedTypes)
                print 'strings', hex(art_object.strings)
                print 'dexFile', hex(art_object.dexFile)

            if object_klass_name=="Ljava/lang/reflect/ArtField;":
                print 'declaringClass', hex(art_object.declaringClass)
                print 'accessFlags', hex(art_object.accessFlags)
                print 'fieldDexIndex', hex(art_object.fieldDexIndex)
                print 'offset', hex(art_object.offset)

            if object_klass_name=="Ljava/lang/reflect/ArtMethod;":
                print 'declaringClass', hex(art_object.declaringClass)
                print 'dexCacheResolvedMethods',
```

```
hex(art_object.dexCacheResolvedMethods)
                print 'dexCacheResolvedTypes', hex(art_object.dexCacheResolvedTypes)
                print 'dexCacheStrings', hex(art_object.dexCacheStrings)
                print 'entryPointFromInterpreter', art_object.entryPointFromInterpreter
                print 'entryPointFromJni', art_object.entryPointFromJni
                print 'entryPointFromQuickCompiledCode',
art_object.entryPointFromQuickCompiledCode
                print 'gcMap', art_object.gcMap
                print 'accessFlags', art_object.accessFlags
                print 'dexCodeItemOffset', art_object.dexCodeItemOffset
                print 'dexMethodIndex', art_object.dexMethodIndex
                print 'methodIndex', art_object.methodIndex

            if object_klass_name=="Ljava/lang/StackTraceElement;":
                print 'declaringClass', hex(art_object.declaringClass),''.join(str(c)
for c in art_object.declaringClass.name.array)
                print 'fileName', hex(art_object.fileName),''.join(str(c) for c in
art_object.fileName.array)
                print 'methodName', hex(art_object.methodName),''.join(str(c) for c in
art_object.methodName.array)
                print 'lineNumber', hex(art_object.lineNumber)

            if object_klass_name=="Ljava/lang/ref/Reference;":
                print 'class layout not implemented yet'
            if object_klass_name=="Ljava/lang/reflect/Proxy;":
                print 'class layout not implemented yet'
            if object_klass_name=="Ljava/lang/ClassLoader;":
                print 'class layout not implemented yet'
            if object_klass_name=="Ljava/lang/Throwable;":
                print 'class layout not implemented yet'
            if object_klass_name=="Ljava/lang/ClassNotFoundException;":
                print 'class layout not implemented yet'
            if object_klass_name=="Z":
                print 'class layout not implemented yet'
            if object_klass_name=="B":
                print 'class layout not implemented yet'
            if object_klass_name=="C":
                print 'class layout not implemented yet'
            if object_klass_name=="D":
                print 'class layout not implemented yet'
            if object_klass_name=="F":
                print 'class layout not implemented yet'
            if object_klass_name=="I":
                print hex(obj.Object('int', vm = self.vm, offset = obj_address+8))
                #print 'class layout not implemented yet'
            if object_klass_name=="J":
                print obj.Object('long long', vm = self.vm, offset = obj_address+8)
                print 'class layout not implemented yet'
            if object_klass_name=="S":
                print 'class layout not implemented yet'
            if object_klass_name=="V":
                print 'class layout not implemented yet'
            if object_klass_name=="[Z":
                print 'class layout not implemented yet'
            if object_klass_name=="[B":
                print 'class layout not implemented yet'
            if object_klass_name=="[D":
                print 'class layout not implemented yet'
            if object_klass_name=="[F":
                print 'class layout not implemented yet'
            if object_klass_name=="[I":
                print 'class layout not implemented yet'
            if object_klass_name=="[J":
                print 'class layout not implemented yet'
            if object_klass_name=="[S":
                print 'class layout not implemented yet'
            if object_klass_name=="[Ljava/lang/StackTraceElement;":
                print 'class layout not implemented yet'

    def calculate(self):
        print 'Object Address:', hex(self._config.OBJECT_ADDRESS)
        print 'Address for IMAGE BOOT:',hex(self._config.BOOT_ART_ADDRESS)

        if not self._config.PID:
```

```
            print "This plugin requires a PID to be given via the '-p' switch"
            return
        #Get process maps for VMA
        #*********************************************************************
        proc_maps = linux_proc_maps.linux_proc_maps(self._config).calculate()

        for task, vma in proc_maps:
            if str(task.pid) == str(self._config.PID):
                break
            continue
        self.task = task
        self.vm = task.get_process_address_space()


        #Loading root classes references
        #*********************************************************************
        bootArt = obj.Object('ArtHeader', vm = self.vm, offset =
self._config.BOOT_ART_ADDRESS)
        imageRoots = obj.Object('[Ljava/lang/Object;', vm = self.vm, offset =
int(bootArt.image_roots))
        kClassRoots = obj.Object('[Ljava/lang/Object;', vm = self.vm, offset =
int(imageRoots.values[7]))
        #print bootArt.magic, bootArt.version
        #print "class_root_length", kClassRoots.values[0], kClassRoots.values[1],
kClassRoots.values[2]
        class_count = 0;

        for class_descriptor in art_extract_object_data.class_roots_descriptors:
            object_klass_name = kClassRoots.values[class_count]
            class_count = class_count + 1

art_extract_object_data.class_root_table[int(object_klass_name)]=class_descriptor

        #print "Class Root table",art_extract_object_data.class_root_table


        #Load object header for identifying Object Class
        target_art_class = obj.Object('Ljava/lang/Object;', vm = self.vm, offset =
self._config.OBJECT_ADDRESS)
        if int(target_art_class.klass) == -1:
            print 'Object not loaded in memory'
            return


        # Verify if it's class is a ROOT class
        object_class_pointer = target_art_class
        if not int(object_class_pointer.klass) in
art_extract_object_data.class_root_table.keys(): #root os not root? this is the
question.
            # if class name parameter, load class memory layout
            if not self._config.CLASS_NAME == None:
                print self._config.CLASS_NAME
                if self._config.CLASS_NAME == "pointer":
                    print hex(target_art_class.klass)
                    sys.exit(0)
                else:
                    object_klass_name = self._config.CLASS_NAME
            else:
                print "(classtype not root)"
                currentSlot = self.vm.zread(self._config.OBJECT_ADDRESS, 160)
                for offset, hexchars, chars in utils.Hexdump(bytes(currentSlot)):
                    print "{1:<48}  {2}".format(0, hexchars, ''.join(chars))
                sys.exit(0)
        else: # root class, known memory layout based on root address.
            print "(classtype is root)"
            object_klass_name =
art_extract_object_data.class_root_table[int(object_class_pointer.klass)]

        print "target_art_class",hex(target_art_class.klass)
        #print "object class pointer",hex(object class pointer), object klass name
        self.retriveObjectData(self._config.OBJECT_ADDRESS, object_klass_name)
```

```
    def render_text(self, outfd, data):
        self.table_header(outfd, [("------------------------------", "32")])
        sys.exit(0)
#        for stringObject in data:
 #           self.table row(outfd, hex(int(stringObject.java lang String_)),
  #                                 hex(int(stringObject.offset )),
   #                                hex(int(stringObject.count )),
    #                               ''.join(str(c) for c in stringObject.array_))
```

C.6 ART_EXTRACT_IMAGE_DATA.PY

```
"""
@author:       Alberto Magno
@contact:      alberto.magno@gmail.com
"""

import volatility.obj as obj
import volatility.plugins.linux.flags as linux_flags
import volatility.plugins.linux.common as linux_common
import volatility.plugins.linux.proc_maps as linux_proc_maps
import volatility.scan as scan
import sys, traceback

class art_extract_image_data(linux_common.AbstractLinuxCommand):
    class_root_table = {};

    class_roots_descriptors = ["Ljava/lang/Class;",
                                    "Ljava/lang/Object;",
                                    "[Ljava/lang/Class;",
                                    "[Ljava/lang/Object;",
                                    "Ljava/lang/String;",
                                    "Ljava/lang/DexCache;",
                                    "Ljava/lang/ref/Reference;",
                                    "Ljava/lang/reflect/ArtField;",
                                    "Ljava/lang/reflect/ArtMethod;",
                                    "Ljava/lang/reflect/Proxy;",
                                    "[Ljava/lang/String;",
                                    "[Ljava/lang/reflect/ArtField;",
                                    "[Ljava/lang/reflect/ArtMethod;",
                                    "Ljava/lang/ClassLoader;",
                                    "Ljava/lang/Throwable;",
                                    "Ljava/lang/ClassNotFoundException;",
                                    "Ljava/lang/StackTraceElement;",
                                    "Z",
                                    "B",
                                    "C",
                                    "D",
                                    "F",
                                    "I",
                                    "J",
                                    "S",
                                    "V",
                                    "[Z",
                                    "[B",
                                    "[C",
                                    "[D",
                                    "[F",
                                    "[I",
                                    "[J",
                                    "[S",
                                    "[Ljava/lang/StackTraceElement;"
                                    ];


    def __init__(self, config, *args, **kwargs):
        linux_common.AbstractLinuxCommand.__init__(self, config, *args, **kwargs)
        config.add_option('PID', short_option = 'p', default = None, help = 'Operate on
this Process ID', action = 'store', type = 'str')

    def calculate(self):
        mytask = None
        proc_maps = linux_proc_maps.linux_proc_maps(self._config).calculate()
        for task, vma in proc_maps:
            if not vma.vm_file:
                continue
            if not linux_common.get_path(task, vma.vm_file) == "/data/dalvik-
cache/arm/system@framework@boot.art":
```

```
                continue
            print task.comm
            if not (vma.vm_flags & 0x00000001 and vma.vm_flags & 0x00000002 and not
vma.vm_flags & 0x00000004):
                continue
            mytask = task
            break
        proc_as = mytask.get_process_address_space()
        bootArt = None
        returnData=[]

        bootArt = obj.Object('ArtHeader', vm = proc_as, offset = vma.vm_start)
        imageRoots = obj.Object('[Ljava/lang/Object;', vm = proc_as, offset =
int(bootArt.image_roots))
        kClassRoots = obj.Object('[Ljava/lang/Object;', vm =
task.get_process_address_space(), offset = int(imageRoots.values[7]))
        class_count = 0;
        class_roots = []
        for class_descriptor in art_header_extract_boot_data.class_roots_descriptors:
            object_klass_name = kClassRoots.values[class_count]
            class_count = class_count + 1

art_header_extract_boot_data.class_root_table[int(object_klass_name)]=class_descriptor
            class_roots.append((hex(class_count),class_descriptor,
hex(object_klass_name)))

        if bootArt.magic==['a','r','t','\x0A']:
            returnData.append(hex(int(vma.vm_start)))
            returnData.append(hex(int(vma.vm_end)))
            returnData.append(hex(int(bootArt.image_begin)))
            returnData.append(hex(int(bootArt.image_size)))
            returnData.append(hex(int(bootArt.image_bitmap_offset)))
            returnData.append(hex(int(bootArt.image_bitmap_size)))
            returnData.append(hex(int(bootArt.oat_checksum)))
            returnData.append(hex(int(bootArt.oat_file_begin)))
            returnData.append(hex(int(bootArt.oat_data_begin)))
            returnData.append(hex(int(bootArt.oat_data_end)))
            returnData.append(hex(int(bootArt.oat_file_end)))
            returnData.append(hex(int(bootArt.patch_delta)))
            returnData.append(hex(int(bootArt.image_roots)))
            returnData.append(hex(int(imageRoots.values[0])))
            returnData.append(hex(int(imageRoots.values[1])))
            returnData.append(hex(int(imageRoots.values[2])))
            returnData.append(hex(int(imageRoots.values[3])))
            returnData.append(hex(int(imageRoots.values[4])))
            returnData.append(hex(int(imageRoots.values[5])))
            returnData.append(hex(int(imageRoots.values[6])))
            returnData.append(hex(int(imageRoots.values[7])))
            returnData.append(class_roots)
            returnData.append(hex(int(bootArt.compile_pic)))
            return returnData

    def render_text(self, outfd, data):
        outfd.write("\nBoot.art\n")
        outfd.write("------------------------\n")
        outfd.write("initial_offset:"+data[0]+"\n")
        outfd.write("end_offset:"+data[1]+"\n")
        outfd.write("\nOatHeader\n")
        outfd.write("------------------------\n")
        outfd.write("image_begin:"+data[2]+"\n")
        outfd.write("image_size:"+data[3]+"\n")
        outfd.write("image_bitmap_offset:"+data[4]+"\n")
        outfd.write("image_bitmap_size:"+data[5]+"\n")
        outfd.write("oat_checksum:"+data[6]+"\n")
        outfd.write("oat_file_begin:"+data[7]+"\n")
        outfd.write("oat_data_begin:"+data[8]+"\n")
        outfd.write("oat_data_end:"+data[9]+"\n")
        outfd.write("oat_file_end:"+data[10]+"\n")
        outfd.write("patch_delta:"+data[11]+"\n")
        outfd.write("image_roots:"+data[12]+"\n")
        outfd.write("    kResolutionMethod:"+data[13]+"\n")
        outfd.write("    kImtConflictMethod:"+data[14]+"\n")
        outfd.write("    kDefaultImt:"+data[15]+"\n")
        outfd.write("    kCalleeSaveMethod:"+data[16]+"\n")
```

```
        outfd.write("    kRefsOnlySaveMethod:"+data[17]+"\n")
        outfd.write("    kRefsAndArgsSaveMethod:"+data[18]+"\n")
        outfd.write("    kDexCaches:"+data[19]+"\n")
        outfd.write("    kClassRoots:"+data[20]+"\n")
        for class_root in data[21]:
            outfd.write("        "+class_root[0]+" "+class_root[1]+"
"+class_root[2]+"\n")
        outfd.write("compile_pic:"+data[22]+"\n")
```

C.7 ART_EXTRACT_OAT_DATA.PY

```
"""
@author:       Alberto Magno
@contact:      alberto.magno@gmail.com
"""

import volatility.obj as obj
import volatility.plugins.linux.flags as linux_flags
import volatility.plugins.linux.common as linux_common
import volatility.plugins.linux.proc_maps as linux_proc_maps
import volatility.plugins.linux.dalvik as dalvik
import volatility.scan as scan
import sys, traceback
import volatility.plugins.linux.pslist as linux_pslist
import struct, string, os.path, pickle


class art_extract_oat_data(linux_common.AbstractLinuxCommand):


    def __init__(self, config, *args, **kwargs):
        linux_common.AbstractLinuxCommand.__init__(self, config, *args, **kwargs)
        dalvik.register_option_PID(config)
        config.add_option('OAT_DATA_BEGIN_OFFSET', short_option = 'o', default =
'0xa5a74000',
                          help = 'This is the offset (in hex) where the oat data can be found
based on where oat file is mapped in the process',
                          action = 'store', type = 'str')
        config.add_option('DEX_CLASS_DEF_INDEX', short_option = 'c', default = '0x1394',
                          help = 'ClassDef index in dex file.',
                          action = 'store', type = 'str')
        config.add_option('DEX_TYPE_INDEX', short_option = 't', default = '0x1810',
                          help = 'Type index in dex file',
                          action = 'store', type = 'str')
        config.add_option('SEARCH_CLASS_NAME', short_option = 'n', default = '
Lcom/whatsapp/protocol/l',
                          help = 'Type index in dex file',
                          action = 'store', type = 'str')

        self.map_type = {0x0:"header_item",
                         0x1:"string_id_item",
                         0x2:"type_id_item",
                         0x3:"proto_id_item",
                         0x4:"field_id_item",
                         0x5:"method_id_item",
                         0x6:"class_def_item",
                         0x1000:"map_list",
                         0x1001:"type_list",
                         0x1002:"annotation_set_ref_list",
                         0x1003:"annotation_set_item",
                         0x2000:"class_data_item",
                         0x2001:"code_item",
                         0x2002:"string_data_item",
                         0x2003:"debug_info_item",
                         0x2004:"annotation_item",
                         0x2005:"encoded_array_item",
                         0x2006:"annotations_directory_item"}


    def calculate(self):
        class_found = 0
        oat_data = 0;
        offset_oat_data_begin = int(self._config.OAT_DATA_BEGIN_OFFSET,16)
        dexClassDefIndex = self._config.DEX_CLASS_DEF_INDEX
        dexTypeIndex = self._config.DEX_TYPE_INDEX
```

```
        searchClassName = self._config.SEARCH_CLASS_NAME.strip()
        file_prefix =
str(hex(offset_oat_data_begin))+'_pid'+str(self._config.PID)+"_"+'dex.'
        self.stringlist_filename = file_prefix+'string.list'
        self.typelist_filename = file_prefix+'type.list'
        self.protolist_filename = file_prefix+'proto.list'
        self.fieldlist_filename = file_prefix+'field.list'
        self.methodlist_filename = file_prefix+'method.list'
        self.classlist_filename = file_prefix+'class.list'

        #print 'Searching class type of index',hex(dexTypeIndex),'at
position',hex(dexClassDefIndex)
        print 'Searching class',searchClassName,'definition'
        if not self._config.PID:
            print "This plugin requires a PID to be given via the '-p' switch"
            return
        proc_maps = linux_proc_maps.linux_proc_maps(self._config).calculate()

        for task, vma in proc_maps:
            if str(task.pid) == str(self._config.PID):
                # print str(task.pid),str(self._config.PID)
                break
            continue

        elfOatHeader = obj.Object('elf_hdr', offset = offset_oat_data_begin, vm =
task.get_process_address_space())
        if not elfOatHeader == None:
            print '\nELF header'
            print "-----------------------\n"
            print 'e_ident',elfOatHeader.e_ident
            print 'e_type',elfOatHeader.e_type
            print 'e_machine',hex(elfOatHeader.e_machine)
            print 'e_version',hex(elfOatHeader.e_version)
            print 'e_entry',hex(elfOatHeader.e_entry)
            print 'e_phoff',hex(elfOatHeader.e_phoff)
            print 'e_shoff',hex(elfOatHeader.e_shoff)
            print 'e_flags',hex(elfOatHeader.e_flags)
            print 'e_ehsize',hex(elfOatHeader.e_ehsize)
            print 'e_phentsize',hex(elfOatHeader.e_phentsize)
            print 'e_phnum',hex(elfOatHeader.e_phnum)
            print 'e_shentsize',hex(elfOatHeader.e_shentsize)
            print 'e_shnum',hex(elfOatHeader.e_shnum)
            print 'e_shstrndx',hex(elfOatHeader.e_shstrndx)
            print '\nProgram headers'
            print "-----------------------\n"
            for phdr in elfOatHeader.program_headers():
                print
'type:',phdr.p_type,'offset:',hex(phdr.p_offset),'vaddr:',hex(phdr.p_vaddr),'p_paddr:',hex(p
hdr.p_paddr),
'p_filesz:',hex(phdr.p_filesz),'memsz:',hex(phdr.p_memsz),'flags:',hex(phdr.p_flags),'align:
',hex(phdr.p_align)
                if (str(phdr.p_type)=='PT_DYNAMIC'):
                    print '\nDynamic Sections'
                    print "-----------------------\n"
                    for dyn in phdr.dynamic_sections():
                        print 'tag:',dyn.d_tag,'ptr',dyn.d_ptr
                        continue
                continue
            oat_header_offset = 0
            print '\nSection headers'
            print "-----------------------\n"
            for shdr in elfOatHeader.section_headers():
                shstr = elfOatHeader.section_header(elfOatHeader.e_shstrndx)
                sname = obj.Object('String', vm = task.get_process_address_space(), offset =
offset_oat_data_begin+shstr.sh_offset+shdr.sh_name,length=8)
                print hex(offset_oat_data_begin+shstr.sh_offset+shdr.sh_name),'name:',sname
                print 'name:',hex(shdr.sh_name),
'type:',hex(shdr.sh_type),'flags:',hex(shdr.sh_flags),'addr:',hex(shdr.sh_addr),'offset:',he
x(shdr.sh_offset),'size:',hex(shdr.sh_size),'link:',hex(shdr.sh_link),
'info:',hex(shdr.sh_info),'addralign:',hex(shdr.sh_addralign),'entsize:',hex(shdr.sh_entsize
)
                if (shdr.sh_type == 11 ): #read dynsym data
                    shstr_dynsym = elfOatHeader.section_header(shdr.sh_link)
                    for index in range(0,shdr.sh_size/shdr.sh_entsize):
```

```
                        sym = obj.Object('elf32_sym', vm = task.get_process_address_space(),
offset = offset_oat_data_begin+shdr.sh_offset+index*shdr.sh_entsize)
                        name = obj.Object('String', length=16 ,vm =
task.get_process_address_space(), offset =
offset_oat_data_begin+shstr_dynsym.sh_offset+shdr.sh_name+sym.st_name-1)
                        print '
name:',str(sym.st_name)+'('+name+')','value',hex(sym.st_value),'size',hex(sym.st_size),'info
',hex(sym.st_info),'other',hex(sym.st_other),'shndx',hex(sym.st_shndx)
                        print '--->',hex(sym.st_value),name.strip()
                        if name.strip()=='oatdata': #get oat data offset inside ELF
                         oat_header_offset = sym.st_value
                continue

            print '\nSymbols'
            print "----------------------\n"
            for sym in elfOatHeader.symbols():
                print 'name:',hex(sym.st_name),
'value:',hex(sym.st_value),'size:',hex(sym.st_size),'info:',hex(sym.st_info),'other:',hex(sy
m.st_other),'index:',hex(sym.st_shndx)
                continue
            oat_data_pointer = offset_oat_data_begin+oat_header_offset
        else:
                oat_data_pointer = offset_oat_data_begin+0x1000 #presuming 0x1000 zeroed
header
        print oat_data_pointer
        bootArt = None
        retorno=[]
        bootOat = obj.Object('OatHeader', vm = task.get_process_address_space(), offset =
oat_data_pointer)
        print '\nDex header'
        print "----------------------\n"

        oatDexHeader = obj.Object('OatDexHeader', vm = task.get_process_address_space(),
offset = oat_data_pointer+bootOat.size()+bootOat.key_value_store_size)
        print 'dex_file_location_size', hex(oatDexHeader.dex_file_location_size)
        print 'dex_file_location_data', ''.join(str(c) for c in
oatDexHeader.dex_file_location_data)
        print 'dex_file_location_checksum', hex(oatDexHeader.dex_file_location_checksum())
        print 'dex_file_pointer', hex(oatDexHeader.dex_file_pointer())
        print 'classes_offsets'
        count_class = 0
        for offset in
oatDexHeader.classes_offsets(bootOat.size()+bootOat.key_value_store_size):
            print hex(count_class), '
offset:',hex(offset),'location',hex(oat_data_pointer+oatDexHeader.dex_file_pointer()+offset)
            count_class +=1
            continue

        # ------------------------------------------------------------------------------------
-------------------------
        dexCount = 1
        dex_pointer = oat_data_pointer+oatDexHeader.dex_file_pointer()
        while dexCount <= bootOat.dex_file_count:


            dexHeader = obj.Object('DexHeader', offset = dex_pointer, vm =
task.get_process_address_space())
            print ""
            print '----------------------------------------------------------------------------
--------------------------------------------'
            print "DEX",dexCount,"of",bootOat.dex_file_count
            print '----------------------------------------------------------------------------
--------------------------------------------'
            print 'dex_file_magic',''.join(str(c) for c in dexHeader.dex_file_magic)
            print 'checksum',hex(dexHeader.checksum)
            print 'signature', hex(int(''.join(str(c) for c in dexHeader.signature)))
            print 'file_size', hex(dexHeader.file_size)
            print 'header_size', hex(dexHeader.header_size)
            print 'endian_tag', hex(dexHeader.endian_tag)
            print 'link_size', hex(dexHeader.link_size)
            print 'link_off', hex(dexHeader.link_off)
            print 'map_off', hex(dexHeader.map_off)
            print 'string_ids_size', hex(dexHeader.string_ids_size)
            print 'string_ids_off', hex(dexHeader.string_ids_off)
```

```
            print 'type_ids_size', hex(dexHeader.type_ids_size)
            print 'type_ids_off', hex(dexHeader.type_ids_off)
            print 'proto_ids_size', hex(dexHeader.proto_ids_size)
            print 'proto_ids_off', hex(dexHeader.proto_ids_off)
            print 'field_ids_size', hex(dexHeader.field_ids_size)
            print 'field_ids_off', hex(dexHeader.field_ids_off)
            print 'method_ids_size', hex(dexHeader.method_ids_size)
            print 'method_ids_off', hex(dexHeader.method_ids_off)
            print 'class_defs_size', hex(dexHeader.class_defs_size)
            print 'class_defs_off', hex(dexHeader.class_defs_off)
            print 'data_size', hex(dexHeader.data_size)
            print 'data_off', hex(dexHeader.data_off)

            print '\nDex map'
            print "-----------------------\n"
            dexMap = obj.Object('DexMap', offset = dex_pointer+dexHeader.map_off, vm =
task.get_process_address_space())
            print 'size',dexMap.count
            for dexMapItem in dexMap.values:
                print
self.map_type[int(dexMapItem.type)],dexMapItem.sizeof,hex(dexMapItem.offset)
    #             print dexMapItem.type," ",dexMapItem.size," ",hex(dexMapItem.offset)
                continue


            print '\nDex strings'
            print "-----------------------\n"
            if not os.path.isfile(self.stringlist_filename):
                string_count = dexMap.values[1].sizeof
                string_table_off = dexMap.values[1].offset
                print "string num:",string_count," offset:",string_table_off
                stringDex = {}
                count = 0
                while(count < string_count):
                    #print count, #hex(count),'(',count,'/',string count,')',
                    stroffset  = obj.Object('unsigned int', offset =
dex_pointer+string_table_off+count*4, vm = task.get_process_address_space())
                    strlen,readByte = self.DecUnsignedLEB128(offset = stroffset+dex_pointer,
vm = task.get_process_address_space())
                    if(strlen == 0) or (strlen > len(searchClassName)+2):
                        stringDex[count] = ""
                        count += 1
                        continue
                    read_string = obj.Object('String' , encoding='utf8', length= strlen,
offset = dex_pointer+stroffset+readByte, vm = task.get_process_address_space())
                    if not (read_string == None):
                        stringDex[count] = str(read_string)
                    else:
                        stringDex[count] = 'Could not load string in this position',
hex(dex_pointer+stroffset+readByte)
                    #print hex(count), filter(lambda x: x in string.printable,
stringDex[count])
                    if searchClassName in stringDex[count]:
                        print 'Found string:',searchClassName, 'at', hex(count)
                        #break
                    count += 1
                with open(self.stringlist_filename, 'wb') as f:
                    pickle.dump(stringDex, f)
            else:
                print 'Loading',self.stringlist_filename
                with open(self.stringlist_filename, 'rb') as f:
                    stringDex = pickle.load(f)

            print '\nDex types'
            print "-----------------------\n"
            if not os.path.isfile(self.typelist_filename):
                type_count = dexMap.values[2].sizeof
                type_table_off = dexMap.values[2].offset
                print "type num:",type_count," offset:",type_table_off
                typeDex = {}
                count = 0
                while(count < type_count):
                    proc_as = task.get_process_address_space()
                    type_data = bytes(proc_as.zread(dex_pointer+type_table_off+count*4, 4))
```

```
                    type = struct.unpack("i",type_data)[0]
                    typeDex[count] = stringDex[type]
                    if searchClassName in stringDex[type]:
                        print count, "Type",hex(type),stringDex[type]
                        #break
                    #print hex(count), type,stringDex[type]
                    count+=1
                with open(self.typelist_filename, 'wb') as f:
                    pickle.dump(typeDex, f)
            else:
                print 'Loading',self.typelist_filename
                with open(self.typelist_filename, 'rb') as f:
                    typeDex = pickle.load(f)

            print '\nDex protos'
            print "----------------------\n"
            if not os.path.isfile(self.protolist_filename):
                proto_count = dexMap.values[3].sizeof
                proto_table_off = dexMap.values[3].offset
                print "proto num:",proto_count," offset:",proto_table_off
                protoDex = {}
                count = 0
                while(count < proto_count):
                    proc_as = task.get_process_address_space()
                    proto_data = bytes(proc_as.zread(dex_pointer+proto_table_off+count*12,
12))

shorty_idx,return_type_idx,parameters_off=struct.unpack("III",proto_data)
                    proto = ""
                    if return_type_idx in typeDex.keys() and shorty_idx in stringDex.keys():
                        proto = hex(count)+"proto:"+stringDex[shorty_idx]+" |
"+typeDex[return_type_idx]+" | " #stringDex[shorty_idx]
                        #print hex(count),"proto:",stringDex[shorty_idx]," |
",typeDex[return_type_idx]," | ",
                    else:
                        proto = hex(count)+"proto:"+hex(shorty_idx)+" |
"+hex(return_type_idx)+" | "
                        #print hex(count),"proto:",shorty_idx," | ",return_type_idx," | ",

                    if parameters_off == 0:
                        proto = proto.join( 'index '+ hex(shorty_idx)+' not loaded in string
list')
                        count+=1
                        continue

                    parameter_data = bytes(proc_as.zread(dex_pointer+parameters_off, 4))
                    parnum = struct.unpack("I",parameter_data)[0]
                    i = 0
                    while(i < parnum):
                        param = bytes(proc_as.zread(dex_pointer+parameters_off+i*2, 2))
                        idx = struct.unpack("H",param)[0]
                        if idx in typeDex.keys():
                            #print typeDex[idx],count
                            proto = proto+'|'+(typeDex[idx])
                        else:
                            #print hex(idx)
                            proto = proto+'|'+hex(idx)
                        i+=1
                    protoDex[count] = proto
                    count+=1
                with open(self.protolist_filename, 'wb') as f:
                    pickle.dump(protoDex, f)
            else:
                print 'Loading',self.protolist_filename
                with open(self.protolist_filename, 'rb') as f:
                    protoDex = pickle.load(f)


            print '\nDex fields'
            print "----------------------\n"
            if not os.path.isfile(self.fieldlist_filename):
                field_count = dexMap.values[4].sizeof
                field_table_off = dexMap.values[4].offset
                print "field num:",field_count," offset:",field_table_off
```

```
                fieldDex = {}
                count = 0
                while(count < field_count):
                    proc_as = task.get_process_address_space()
                    field_data = bytes(proc_as.zread(dex_pointer+field_table_off+count*8,
8))
                    class_idx,type_idx,name_idx = struct.unpack("HHI",field_data)
                    #print count, hex(class_idx)," | ",hex(type_idx)," | ",hex(name_idx),
                    if name_idx in stringDex.keys():
                        #print stringDex[name_idx]
                        fieldDex[count] =
(typeDex[class_idx],typeDex[type_idx],stringDex[name_idx])
                    else:
                        #print '(name not in string list)'
                        fieldDex[count] = (typeDex[class_idx],typeDex[type_idx],'(name not
in string list)')
                    count+=1
                with open(self.fieldlist_filename, 'wb') as f:
                    pickle.dump(fieldDex, f)
            else:
                print 'Loading',self.fieldlist_filename
                with open(self.fieldlist_filename, 'rb') as f:
                    fieldDex = pickle.load(f)

            print '\nDex methods'
            print "-----------------------\n"
            if not os.path.isfile(self.methodlist_filename):
                method_count = dexMap.values[5].sizeof
                method_table_off = dexMap.values[5].offset
                print "method num:",method_count," offset:",method_table_off
                methodDex = {}
                count = 0
                while(count < method_count):
                    proc_as = task.get_process_address_space()
                    method_data = bytes(proc_as.zread(dex_pointer+method_table_off+count*8,
8))
                    class_idx,proto_idx,name_idx = struct.unpack("HHI",method_data)
                    #print count, hex(class_idx)," | ",hex(proto_idx)," | ",hex(name_idx),
name idx,
                    if name_idx in stringDex.keys() and class_idx in typeDex.keys() and
proto_idx in protoDex.keys():
                        #print stringDex[name_idx]
                        methodDex[count] =
(typeDex[class_idx],protoDex[proto_idx],stringDex[name_idx])
                    else:
                        #print '(name not in string list)'
                        methodDex[count] = (hex(class_idx),hex(proto_idx),hex(name_idx))
                    count+=1
                with open(self.methodlist_filename, 'wb') as f:
                    pickle.dump(methodDex, f)
            else:
                print 'Loading',self.methodlist_filename
                with open(self.methodlist_filename, 'rb') as f:
                    methodDex = pickle.load(f)

            print '\nDex classes'
            print "-----------------------\n"
            if not os.path.isfile(self.classlist_filename):
                print "class def size:",hex(dexHeader.class_defs_size)
                classDex = {}
                count = 0
                while(count < dexHeader.class_defs_size and class_found==0):
                    proc_as = task.get_process_address_space()
                    class_data =
bytes(proc_as.zread(dex_pointer+dexHeader.class_defs_off+count*32, 32))
                    class_idx,\
                    access_flags,\
                    superclass_idx,\
                    interfaces_off,\
                    source_file_idx,\
                    annotations_off,\
                    class_data_off,\
                    static_values_off = struct.unpack("IIIIIIII",class_data)
```

```
                    if class_data_off == 0:# or  not(searchClassName in typeDex[class_idx]):
                        count+=1
                        continue
                    if (class_idx in typeDex.keys()):
                        classDex[count] = typeDex[class_idx]
                    else:
                        classDex[count] = "Class idx",class_idx
                    count+=1

                    #comentar
                    offsetLEB128 = class_data_off + dexHeader.class_defs_off + dex_pointer
                    print "Class offset",offsetLEB128
                    static_fields_size,readBytes = self.DecUnsignedLEB128(offset =
offsetLEB128, vm = task.get_process_address_space())
                    offsetLEB128 +=  readBytes
                    instance_fields_size,readBytes = self.DecUnsignedLEB128(offset =
offsetLEB128, vm = task.get_process_address_space())
                    offsetLEB128 +=  readBytes
                    direct_methods_size,readBytes = self.DecUnsignedLEB128(offset =
offsetLEB128 , vm = task.get_process_address_space())
                    offsetLEB128 +=  readBytes
                    virtual_methods_size,readBytes = self.DecUnsignedLEB128(offset =
offsetLEB128 , vm = task.get_process_address_space())
                    offsetLEB128 +=  readBytes

                    print "-----> static_fields_size",static_fields_size
                    field_idx = 0
                    while(static_fields_size > 0):
                        field_idx_diff,readBytes = self.DecUnsignedLEB128(offset =
offsetLEB128 , vm = task.get_process_address_space())
                        offsetLEB128 +=  readBytes
                        field_pointer = field_idx
                        if field_idx_diff > 0 :
                            field_pointer += field_idx_diff
                        if field_pointer in fieldDex.keys():
                            print fieldDex[field_pointer]
                        else:
                            print field_pointer ,' (not in dex static fields list)'
                        #field idx+=field idx diff
                        field_idx = field_pointer
                        access_flags,readBytes = self.DecUnsignedLEB128(offset =
offsetLEB128 , vm = task.get_process_address_space())
                        offsetLEB128 +=  readBytes
                        static_fields_size-=1

                    field_idx = 0
                    print "-----> instance_fields_size",instance_fields_size
                    while(instance_fields_size > 0):
                        field_idx_diff,readBytes = self.DecUnsignedLEB128(offset =
offsetLEB128 , vm = task.get_process_address_space())
                        offsetLEB128 +=  readBytes
                        field_pointer = (field_idx + field_idx_diff)
                        if  field_pointer in fieldDex.keys():
                            print fieldDex[field_pointer]
                        else:
                            print field_pointer ,' (not in dex instance fields list)'
                        #field idx =field idx diff
                        field_idx = field_pointer
                        access_flags,readBytes = self.DecUnsignedLEB128(offset =
offsetLEB128 , vm = task.get_process_address_space())
                        offsetLEB128 +=  readBytes
                        instance_fields_size-=1

                    method_idx = 0
                    print "-----> direct_methods_size",direct_methods_size
                    while(direct_methods_size > 0):
                        method_idx_diff,readBytes = self.DecUnsignedLEB128(offset =
offsetLEB128 , vm = task.get_process_address_space())
                        offsetLEB128 +=  readBytes
                        method_pointer = method_idx + method_idx_diff
                        if method_pointer in methodDex.keys():
                            print methodDex[method_pointer]
                        else:
                            print (method_pointer) ,' (not in dex direct methods list)'
```

```
                        #method idx+=method idx diff
                        method_idx = method_pointer
                        access_flags,readBytes = self.DecUnsignedLEB128(offset =
offsetLEB128 , vm = task.get_process_address_space())
                        offsetLEB128 +=  readBytes
                        code_off,readBytes = self.DecUnsignedLEB128(offset = offsetLEB128 ,
vm = task.get_process_address_space())
                        offsetLEB128 +=  readBytes
                        direct_methods_size-=1

                    method_idx = 0
                    print "-----> virtual_methods_size",virtual_methods_size
                    while(virtual_methods_size > 0):
                        method_idx_diff,readBytes = self.DecUnsignedLEB128(offset =
offsetLEB128 , vm = task.get_process_address_space())
                        offsetLEB128 +=  readBytes
                        method_pointer = method_idx + method_idx_diff
                        if method_pointer in methodDex.keys():
                            print methodDex[method_pointer]
                        else:
                            print (method_pointer) ,' (not in dex direct methods list)'
                        #method_idx+=method_idx_diff
                        method_idx = method_pointer
                        access_flags,readBytes = self.DecUnsignedLEB128(offset =
offsetLEB128 , vm = task.get_process_address_space())
                        offsetLEB128 +=  readBytes
                        code_off,readBytes = self.DecUnsignedLEB128(offset = offsetLEB128 ,
vm = task.get_process_address_space())
                        offsetLEB128 +=  readBytes
                        virtual_methods_size-=1
                ## discoment
                with open(self.classlist_filename, 'wb') as f:
                    pickle.dump(classDex, f)
            else:
                print 'Loading',self.classlist_filename
                with open(self.classlist_filename, 'rb') as f:
                    classDex = pickle.load(f)

            print ".....END DEX", dexCount

            dexCount = dexCount+1
            print hex(dex_pointer)
            dex_pointer = dex_pointer+dexHeader.file_size
            print hex(dex_pointer)

        if bootOat.magic==['o','a','t','\x0A']:
            retorno.append(hex(int(oat_data_pointer)))
            retorno.append(hex(int(bootOat.instruction_set)))
            retorno.append(hex(int(bootOat.instruction_set_features)))
            retorno.append(hex(int(bootOat.dex_file_count)))
            retorno.append(hex(int(bootOat.executable_offset)))
            retorno.append(hex(int(bootOat.interpreter_to_interpreter_bridge_offset)))
            retorno.append(hex(int(bootOat.interpreter_to_compiled_code_bridge_offset)))
            retorno.append(hex(int(bootOat.jni_dlsym_lookup_offset_)))
            retorno.append(hex(int(bootOat.portable_imt_conflict_trampoline_offset)))
            retorno.append(hex(int(bootOat.portable_resolution_trampoline_offset)))
            retorno.append(hex(int(bootOat.portable_to_interpreter_bridge_offset)))
            retorno.append(hex(int(bootOat.quick_generic_jni_trampoline_offset)))
            retorno.append(hex(int(bootOat.quick_imt_conflict_trampoline_offset)))
            retorno.append(hex(int(bootOat.quick_resolution_trampoline_offset)))
            retorno.append(hex(int(bootOat.quick_to_interpreter_bridge_offset)))
            retorno.append(hex(int(bootOat.image_patch_delta)))
            retorno.append(hex(int(bootOat.image_file_location_oat_checksum)))
            retorno.append(hex(int(bootOat.image_file_location_oat_data_begin)))
            retorno.append(hex(int(bootOat.key_value_store_size)))
            #linhacomando = obj.Object("String", offset = offset_oat_data_begin+0x54, vm =
task.get process address space(), length = bootOat.key value store size)
            #linhacomando2 = obj.Object("String", offset = offset oat data begin+0x54+0x11,
vm = task.get process address space(), length = bootOat.key value store size)
            key_value_store = ''.join(str(c) for c in bootOat.key_value_store)
            #for caracter in bootOat.key_value_store:
             #   key_value_store.join(caracter)
            retorno.append(key_value_store)
```

```
            return retorno

    def DecUnsignedLEB128(self,offset, vm):
            result = struct.unpack("i",bytes(vm.zread(offset, 4)))[0]
            #result = obj.Object('unsigned int', offset, vm)
            readBytes = 1
            rewind = 3
            result = result&0x000000ff
            if(result > 0x7f):
                next = struct.unpack("i",bytes(vm.zread(offset+1, 4)))[0]
                #next = obj.Object('unsigned int', offset+1, vm)
                readBytes += 1
                next = next&0x000000ff
                #print
'*************************************************************************sec level'
                result = (result&0x7f) | (next&0x7f)<<7
                if(next > 0x7f):
                    next = struct.unpack("i",bytes(vm.zread(offset+2, 4)))[0]
                    #next = obj.Object('unsigned int', offset+2, vm)
                    readBytes += 1
                    next = next&0x000000ff
                    result = result | (next&0x7f)<<14
                    #print
'*************************************************************************third level'
                    if(next > 0x7f):
                        next = struct.unpack("i",bytes(vm.zread(offset+3, 4)))[0]
                        #next = obj.Object('unsigned int', offset+3, vm)
                        readBytes += 1
                        next = next&0x000000ff
                        result = result | (next&0x7f)<<21
                        #print
'*************************************************************************fourth level'
                        if(next > 0x7f):
                            next = struct.unpack("i",bytes(vm.zread(offset+4, 4)))[0]
                            #next = next = obj.Object('unsigned int', offset+4, vm)
                            readBytes += 1
                            next = next&0x000000ff
                            result = result | next<<28
                            #print
'*************************************************************************fith level'
            return result,readBytes

    def render_text(self, outfd, data):
        outfd.write("\noat\n")
        outfd.write("----------------------\n")
        outfd.write("initial_offset:"+data[0]+"\n")
        outfd.write("\nOatHeader\n")
        outfd.write("----------------------\n")
        outfd.write("instruction_set:"+data[1]+"\n")
        outfd.write("instruction_set_features:"+data[2]+"\n")
        outfd.write("dex_file_count:"+data[3]+"\n")
        outfd.write("executable_offset:"+data[4]+"\n")
        outfd.write("interpreter_to_interpreter_bridge_offset:"+data[5]+"\n")
        outfd.write("interpreter_to_compiled_code_bridge_offset:"+data[6]+"\n")
        outfd.write("jni_dlsym_lookup_offset_:"+data[7]+"\n")
        outfd.write("portable_imt_conflict_trampoline_offset:"+data[8]+"\n")
        outfd.write("portable_resolution_trampoline_offset:"+data[9]+"\n")
        outfd.write("portable_to_interpreter_bridge_offset:"+data[10]+"\n")
        outfd.write("quick_generic_jni_trampoline_offset:"+data[11]+"\n")
        outfd.write("quick_imt_conflict_trampoline_offset:"+data[12]+"\n")
        outfd.write("quick_resolution_trampoline_offset:"+data[13]+"\n")
        outfd.write("quick_to_interpreter_bridge_offset:"+data[14]+"\n")
        outfd.write("image_patch_delta:"+data[15]+"\n")
        outfd.write("image_file_location_oat_checksum:"+data[16]+"\n")
        outfd.write("image_file_location_oat_data_begin:"+data[17]+"\n")
        outfd.write("key_value_store_size:"+data[18]+"\n")
        outfd.write("key_value_store:"+data[19]+"\n")
```

C.8 ART_DEX_CLASS_DATA.PY

```
"""
@author:       Alberto Magno
@contact:      alberto.magno@gmail.com
"""
import volatility.obj as obj
import volatility.plugins.linux.flags as linux_flags
import volatility.plugins.linux.common as linux_common
import volatility.plugins.linux.proc_maps as linux_proc_maps
import volatility.scan as scan
import sys, traceback
import volatility.plugins.linux.pslist as linux_pslist
import struct, string, os.path, pickle


class art_dex_class_data():
    def __init__(self, offset_oat_data_begin, PID):

        file_prefix = str(hex(offset_oat_data_begin))+'_pid'+str(PID)+"_"+'dex.'
        stringlist_filename = file_prefix+'string.list'
        typelist_filename = file_prefix+'type.list'
        protolist_filename = file_prefix+'proto.list'
        fieldlist_filename = file_prefix+'field.list'
        methodlist_filename = file_prefix+'method.list'
        classlist_filename = file_prefix+'class.list'

        print '\nDex strings'
        print "-----------------------\n"
        print 'Loading',stringlist_filename
        with open(stringlist_filename, 'rb') as f:
            self.stringDex = pickle.load(f)

        print '\nDex types'
        print "-----------------------\n"
        print 'Loading',typelist_filename
        with open(typelist_filename, 'rb') as f:
            self.typeDex = pickle.load(f)

        print '\nDex protos'
        print "-----------------------\n"
        print 'Loading',protolist_filename
        with open(protolist_filename, 'rb') as f:
            self.protoDex = pickle.load(f)

        print '\nDex fields'
        print "-----------------------\n"
        print 'Loading',fieldlist_filename
        with open(fieldlist_filename, 'rb') as f:
            self.fieldDex = pickle.load(f)

        print '\nDex methods'
        print "-----------------------\n"
        print 'Loading',methodlist_filename
        with open(methodlist_filename, 'rb') as f:
            self.methodDex = pickle.load(f)

        print '\nDex classes'
        print "-----------------------\n"
        print 'Loading',classlist_filename
        with open(classlist_filename, 'rb') as f:
            self.classDex = pickle.load(f)

    def getDexSring(self,dexIndex):
        if dexIndex in self.stringDex.keys():
            return self.stringDex[dexIndex]
```

```
        else:
            return 'not found'
    def getDexType(self,dexIndex):
        if dexIndex in self.typeDex.keys():
            return self.typeDex[dexIndex]
        else:
            return 'not found'
    def getDexField(self,dexIndex):
        if dexIndex in self.fieldDex.keys():
            return self.fieldDex[int(dexIndex)]
        else:
            return 'not found'
    def getDexProto(self,dexIndex):
        if dexIndex in self.protoDex.keys():
            return self.protoDex[dexIndex]
        else:
            return 'not found'
    def getDexMethod(self,dexIndex):
        if dexIndex in self.methodDex.keys():
            return self.methodDex[dexIndex]
        else:
            return 'not found'
    def getDexClassType(self,dexIndex):
        if dexIndex in self.classDex.keys():
            return self.classDex[dexIndex]
        else:
            return 'not found'
```

C.9 ART_EXTRACT_OAT_DEX.PY

```
"""
@author:       Alberto Magno
@contact:      alberto.magno@gmail.com
"""

import volatility.obj as obj
import volatility.plugins.linux.flags as linux_flags
import volatility.plugins.linux.common as linux_common
import volatility.plugins.linux.proc_maps as linux_proc_maps
import volatility.scan as scan
import sys, traceback
import volatility.plugins.linux.pslist as linux_pslist
import struct, string, os.path, pickle


class art_extract_oat_dex(linux_common.AbstractLinuxCommand):


    def __init__(self, config, *args, **kwargs):
        linux_common.AbstractLinuxCommand.__init__(self, config, *args, **kwargs)
        config.add_option('PID', short_option = 'p', default = None, help = 'Operate on
this Process ID', action = 'store', type = 'str')
        config.add_option('OAT_DATA_BEGIN_OFFSET', short_option = 'o', default =
'0xa5a74000',
                          help = 'This is the offset (in hex) where the oat data can be
found based on where oat file is mapped in the process',
                          action = 'store', type = 'str')
        self._config.add_option('DUMP-DIR', short_option = 'D', default = None, help =
'Output directory', action = 'store', type = 'str')

    def write_dex_file(self, dexCount, dex_addr, dex_end):
        file_path = os.path.join(self._config.DUMP_DIR, "%s.%s.dex.%d.%#8x" %
(hex(int(self._config.OAT_DATA_BEGIN_OFFSET,16)), self._config.PID, dexCount,dex_addr))
        file_contents = self.proc_as.zread(dex_addr, dex_end-dex_addr)

        fd = open(file_path, "wb")
        fd.write(file_contents)
        fd.close()

        return file_path

    def calculate(self):
        class_found = 0
        oat_data = 0;
        offset_oat_data_begin = int(self._config.OAT_DATA_BEGIN_OFFSET,16)
        file_prefix =
str(hex(offset_oat_data_begin))+'_pid'+str(self._config.PID)+"_"+'dex.'

        if not self._config.PID:
            print "This plugin requires a PID to be given via the '-p' switch"
            return
        proc_maps = linux_proc_maps.linux_proc_maps(self._config).calculate()

        for task, vma in proc_maps:
            if str(task.pid) == str(self._config.PID):
                # print str(task.pid),str(self._config.PID)
                break
            continue
        self.proc_as = task.get_process_address_space()

        elfOatHeader = obj.Object('elf_hdr', offset = offset_oat_data_begin, vm =
task.get_process_address_space())
        if not elfOatHeader == None:
            print '\nELF header'
            print "-----------------------\n"
            print 'e_ident',elfOatHeader.e_ident
```

```
            print 'e_type',elfOatHeader.e_type
            print 'e_machine',hex(elfOatHeader.e_machine)
            print 'e_version',hex(elfOatHeader.e_version)
            print 'e_entry',hex(elfOatHeader.e_entry)
            print 'e_phoff',hex(elfOatHeader.e_phoff)
            print 'e_shoff',hex(elfOatHeader.e_shoff)
            print 'e_flags',hex(elfOatHeader.e_flags)
            print 'e_ehsize',hex(elfOatHeader.e_ehsize)
            print 'e_phentsize',hex(elfOatHeader.e_phentsize)
            print 'e_phnum',hex(elfOatHeader.e_phnum)
            print 'e_shentsize',hex(elfOatHeader.e_shentsize)
            print 'e_shnum',hex(elfOatHeader.e_shnum)
            print 'e_shstrndx',hex(elfOatHeader.e_shstrndx)
            print '\nProgram headers'
            print "-----------------------\n"
            for phdr in elfOatHeader.program_headers():
                print
'type:',phdr.p_type,'offset:',hex(phdr.p_offset),'vaddr:',hex(phdr.p_vaddr),'p_paddr:',h
ex(phdr.p_paddr),
'p_filesz:',hex(phdr.p_filesz),'memsz:',hex(phdr.p_memsz),'flags:',hex(phdr.p_flags),'al
ign:',hex(phdr.p_align)
                if (str(phdr.p_type)=='PT_DYNAMIC'):
                    print '\nDynamic Sections'
                    print "-----------------------\n"
                    for dyn in phdr.dynamic_sections():
                        print 'tag:',dyn.d_tag,'ptr',dyn.d_ptr
                        continue
                continue
            oat_header_offset = 0
            print '\nSection headers'
            print "-----------------------\n"
            for shdr in elfOatHeader.section_headers():
                shstr = elfOatHeader.section_header(elfOatHeader.e_shstrndx)
                sname = obj.Object('String', vm = task.get_process_address_space(),
offset = offset_oat_data_begin+shstr.sh_offset+shdr.sh_name,length=8)
                print
hex(offset_oat_data_begin+shstr.sh_offset+shdr.sh_name),'name:',sname
                print 'name:',hex(shdr.sh_name),
'type:',hex(shdr.sh_type),'flags:',hex(shdr.sh_flags),'addr:',hex(shdr.sh_addr),'offset:
',hex(shdr.sh_offset),'size:',hex(shdr.sh_size),'link:',hex(shdr.sh_link),
'info:',hex(shdr.sh_info),'addralign:',hex(shdr.sh_addralign),'entsize:',hex(shdr.sh_ent
size)
                if (shdr.sh_type == 11 ): #read dynsym data
                    shstr_dynsym = elfOatHeader.section_header(shdr.sh_link)
                    for index in range(0,shdr.sh_size/shdr.sh_entsize):
                        sym = obj.Object('elf32_sym', vm =
task.get_process_address_space(), offset =
offset_oat_data_begin+shdr.sh_offset+index*shdr.sh_entsize)
                        name = obj.Object('String', length=16 ,vm =
task.get_process_address_space(), offset =
offset_oat_data_begin+shstr_dynsym.sh_offset+shdr.sh_name+sym.st_name-1)
                        print '
name:',str(sym.st_name)+'('+name+')','value',hex(sym.st_value),'size',hex(sym.st_size),'
info',hex(sym.st_info),'other',hex(sym.st_other),'shndx',hex(sym.st_shndx)
                        print '--->',hex(sym.st_value),name.strip()
                        if name.strip()=='oatdata': #get oat data offset inside ELF
                         oat_header_offset = sym.st_value
                continue

            print '\nSymbols'
            print "-----------------------\n"
            for sym in elfOatHeader.symbols():
                print 'name:',hex(sym.st_name),
'value:',hex(sym.st_value),'size:',hex(sym.st_size),'info:',hex(sym.st_info),'other:',he
x(sym.st_other),'index:',hex(sym.st_shndx)
                continue
            oat_data_pointer = offset_oat_data_begin+oat_header_offset
        else:
            oat_data_pointer = offset_oat_data_begin+0x1000 #presuming 0x1000 zeroed
header (page fault)
        print oat_data_pointer
        bootArt = None
        retorno=[]
        bootOat = obj.Object('OatHeader', vm = task.get_process_address_space(), offset
```

```
= oat_data_pointer)
        print '\nDex header'
        print "-----------------------\n"

        oatDexHeader = obj.Object('OatDexHeader', vm = task.get_process_address_space(),
offset = oat_data_pointer+bootOat.size()+bootOat.key_value_store_size)
        print 'dex_file_location_size', hex(oatDexHeader.dex_file_location_size)
        print 'dex_file_location_data', ''.join(str(c) for c in
oatDexHeader.dex_file_location_data)
        print 'dex_file_location_checksum',
hex(oatDexHeader.dex_file_location_checksum())
        print 'dex_file_pointer', hex(oatDexHeader.dex_file_pointer())
        print 'classes_offsets'
        '''count_class = 0
        for offset in
oatDexHeader.classes_offsets(bootOat.size()+bootOat.key_value_store_size):
            print hex(count_class), '
offset:',hex(offset),'location',hex(oat_data_pointer+oatDexHeader.dex_file_pointer()+off
set)
            count_class +=1
            continue'''

        # ------------------------------------------------------------------------------
-----------------------------
        if bootOat.magic==['o','a','t','\x0A']:
            retorno.append(hex(int(oat_data_pointer)))
            retorno.append(hex(int(bootOat.instruction_set)))
            retorno.append(hex(int(bootOat.instruction_set_features)))
            retorno.append(hex(int(bootOat.dex_file_count)))
            retorno.append(hex(int(bootOat.executable_offset)))
            retorno.append(hex(int(bootOat.interpreter_to_interpreter_bridge_offset)))
            retorno.append(hex(int(bootOat.interpreter_to_compiled_code_bridge_offset)))
            retorno.append(hex(int(bootOat.jni_dlsym_lookup_offset_)))
            retorno.append(hex(int(bootOat.portable_imt_conflict_trampoline_offset)))
            retorno.append(hex(int(bootOat.portable_resolution_trampoline_offset)))
            retorno.append(hex(int(bootOat.portable_to_interpreter_bridge_offset)))
            retorno.append(hex(int(bootOat.quick_generic_jni_trampoline_offset)))
            retorno.append(hex(int(bootOat.quick_imt_conflict_trampoline_offset)))
            retorno.append(hex(int(bootOat.quick_resolution_trampoline_offset)))
            retorno.append(hex(int(bootOat.quick_to_interpreter_bridge_offset)))
            retorno.append(hex(int(bootOat.image_patch_delta)))
            retorno.append(hex(int(bootOat.image_file_location_oat_checksum)))
            retorno.append(hex(int(bootOat.image_file_location_oat_data_begin)))
            retorno.append(hex(int(bootOat.key_value_store_size)))
            key_value_store = ''.join(str(c) for c in bootOat.key_value_store)
            retorno.append(key_value_store)

        dexCount = 1
        dexList=[]
        dex_pointer = oat_data_pointer+oatDexHeader.dex_file_pointer()
        while dexCount <= bootOat.dex_file_count:
            dexHeader = obj.Object('DexHeader', offset = dex_pointer, vm =
task.get_process_address_space())
            print ""
            print '-------------------------------------------------------------------------
-------------------------------------------------'
            print "DEX",dexCount,"of",bootOat.dex_file_count
            print '-------------------------------------------------------------------------
-------------------------------------------------'
            print 'dex_file_magic',''.join(str(c) for c in dexHeader.dex_file_magic)
            print 'checksum',hex(dexHeader.checksum)
            print 'signature', hex(int(''.join(str(c) for c in dexHeader.signature)))
            print 'file_size', hex(dexHeader.file_size)
            print 'header_size', hex(dexHeader.header_size)
            print 'endian_tag', hex(dexHeader.endian_tag)
            print 'link_size', hex(dexHeader.link_size)
            print 'link_off', hex(dexHeader.link_off)
            print 'map_off', hex(dexHeader.map_off)
            print 'string_ids_size', hex(dexHeader.string_ids_size)
            print 'string_ids_off', hex(dexHeader.string_ids_off)
            print 'type_ids_size', hex(dexHeader.type_ids_size)
            print 'type_ids_off', hex(dexHeader.type_ids_off)
            print 'proto_ids_size', hex(dexHeader.proto_ids_size)
            print 'proto_ids_off', hex(dexHeader.proto_ids_off)
```

```
            print 'field_ids_size', hex(dexHeader.field_ids_size)
            print 'field_ids_off', hex(dexHeader.field_ids_off)
            print 'method_ids_size', hex(dexHeader.method_ids_size)
            print 'method_ids_off', hex(dexHeader.method_ids_off)
            print 'class_defs_size', hex(dexHeader.class_defs_size)
            print 'class_defs_off', hex(dexHeader.class_defs_off)
            print 'data_size', hex(dexHeader.data_size)
            print 'data_off', hex(dexHeader.data_off)
            print ".....END DEX", dexCount

            dexCount = dexCount+1
            dex_end = dex_pointer+dexHeader.file_size
            dexList.append((dex_pointer,dex_end))
            dex_pointer = dex_end
        retorno.append(dexList)
        return retorno
    def render_text(self, outfd, data):
        if not self._config.DUMP_DIR:
            debug.error("-D/--dump-dir must given that specifies an existing directory")
        outfd.write("\noat\n")
        outfd.write("----------------------\n")
        outfd.write("initial_offset:"+data[0]+"\n")
        outfd.write("\nOatHeader\n")
        outfd.write("----------------------\n")
        outfd.write("instruction_set:"+data[1]+"\n")
        outfd.write("instruction_set_features:"+data[2]+"\n")
        outfd.write("dex_file_count:"+data[3]+"\n")
        outfd.write("executable_offset:"+data[4]+"\n")
        outfd.write("interpreter_to_interpreter_bridge_offset:"+data[5]+"\n")
        outfd.write("interpreter_to_compiled_code_bridge_offset:"+data[6]+"\n")
        outfd.write("jni_dlsym_lookup_offset_:"+data[7]+"\n")
        outfd.write("portable_imt_conflict_trampoline_offset:"+data[8]+"\n")
        outfd.write("portable_resolution_trampoline_offset:"+data[9]+"\n")
        outfd.write("portable_to_interpreter_bridge_offset:"+data[10]+"\n")
        outfd.write("quick_generic_jni_trampoline_offset:"+data[11]+"\n")
        outfd.write("quick_imt_conflict_trampoline_offset:"+data[12]+"\n")
        outfd.write("quick_resolution_trampoline_offset:"+data[13]+"\n")
        outfd.write("quick_to_interpreter_bridge_offset:"+data[14]+"\n")
        outfd.write("image_patch_delta:"+data[15]+"\n")
        outfd.write("image_file_location_oat_checksum:"+data[16]+"\n")
        outfd.write("image_file_location_oat_data_begin:"+data[17]+"\n")
        outfd.write("key_value_store_size:"+data[18]+"\n")
        outfd.write("key_value_store:"+data[19]+"\n")

        self.table_header(outfd, [("Dex", "3"),
                                  ("Pid", "5"),
                                  ("OAT", "15"),
                                  ("Start", "15"),
                                  ("Size", "15"),
                                  ("Output File", "50")])
        count = 1
        for dex in data[20]:
            file_path = self.write_dex_file(count,dex[0], dex[1])
            self.table_row(outfd, count,
                                  hex(int(self._config.PID,16)),
                                  hex(int(self._config.OAT_DATA_BEGIN_OFFSET,16)),
                                  hex(dex[0]),
                                  hex(dex[1]-dex[0]) ,
                                  file_path)
            count = count + 1
```