UNIVERSIDADE FEDERAL DO MARANHÃO
DEPARTAMENTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

MÁRIO HENRIQUE BRAGA SANTOS

APRIMORAMENTO DO FRAMEWORK DE HONEYPOT VIRTUAL PARA ANDROID: Framework Labsac 2.0

São Luís
2018

MÁRIO HENRIQUE BRAGA SANTOS

APRIMORAMENTO DO FRAMEWORK DE HONEYPOT VIRTUAL PARA ANDROID: Framework Labsac 2.0

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal do Maranhão, como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Mário Antonio Meireles Teixeira

São Luís

2018

Ficha gerada por meio do SIGAA/Biblioteca com dados fornecidos pelo(a) autor(a).
Núcleo Integrado de Bibliotecas/UFMA

Santos, Mário Henrique Braga.
Aprimoramento do framework de Honeypot virtual para Android : Framework Labsac 2.0 / Mário Henrique Braga Santos. - 2018.
57 f.

Orientador(a): Mário Antonio Meireles Teixeira.
Monografia (Graduação) - Curso de Ciência da Computação, Universidade Federal do Maranhão, São Luís, 2018.

1. Dispositivos móveis. 2. Honeypot. 3. Plataforma Android. 4. Segurança de rede. I. Teixeira, Mário Antonio Meireles. II. Título.

MÁRIO HENRIQUE BRAGA SANTOS

APRIMORAMENTO DO FRAMEWORK DE HONEYPOT VIRTUAL PARA ANDROID: Framework Labsac 2.0

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal do Maranhão, como parte dos requisitos necessários para obtenção do grau de Bacharel em Ciência da Computação.

Aprovada em: 23 / 10 / 2018

BANCA EXAMINADORA

Prof. Dr. Mário Antonio Meireles Teixeira (Orientador)
Doutor em Ciência da Computação
Universidade Federal do Maranhão

Prof. Dr. Samyr Béliche Vale
Doutor em Ciência da Computação
Université dAngers - França

Prof. Mestre Higo Felipe Silva Pires
Mestre em Engenharia Elétrica
Universidade Federal do Maranhão

À minha mãe Francinete, ao meu pai Mário, aos meus irmãos Gustavo e Paula por toda luta que tiveram e pelos ensinamentos e conselhos dados.

AGRADECIMENTOS

À minha mãe, Francinete Braga Santos, pela grande paciência por todos esses anos no apoio e orientações, por também dar exemplo de dedicação aos estudos e dar dicas na conclusão dessa monografia.

Ao meu pai, Mário dos Santos, pelos muitos conselhos e conversas longas e puxões de orelha (foram mais puxões de orelha), por sempre querer o melhor para todos os filhos, para ser sempre melhor.

Aos meus irmãos, Gustavo e Paula Francinete, pelo apoio, conselhos e divisão de experiências, e porque não, pelas brigas para dividir a responsabilidade em casa.

À minha família no geral, por todos esses anos pegando no meu pé para terminar o curso e perguntando quando iria terminar.

Um agradecimento muito especial à Andréia Monteiro Carvalho, que me apoiou, incentivou, me puxou a orelha e dedicou uma boa parte de seu tempo para me ajudar científica e psicologicamente na conclusão desta odisseia que é minha monografia.

Aos meus amigos Leonardo Maciel e Wellington Jorge, pelo companheirismo de muitos anos, pelo apoio, pelas diversas conversas interessantes e diversão.

À equipe de Taekwondo da UFMA, que a frente estavam os professores Itânio Soares, Elaynne Silva e Tomaz Soares por me proporcionarem o autoconhecimento, o melhoramento do corpo e mente por meio da filosofia dessa arte marcial centenária.

À minha amiga do Taekwondo, Railde Paula, primeira pessoa do grupo com quem falei, e que se tornou depois uma pessoa da qual tenho muito apreço, uma pessoa alegre e comprometida, pelo grande incentivo na escrita deste trabalho.

Aos colegas do Laboratório de Sistemas e Arquiteturas Computacionais – LABSAC que serviram como exemplo, como o Leonardo Melo, Johnatan Iury, Cláudio Aroucha, Higo Felipe, Steve Ataky. Destacando o Vladimir Oliveira, o qual ajudei a construir seu trabalho e do qual me inspirei.

Ao professor Ph.D. Zair Abdolouahab (*in memoriam*) pela grande demonstração de conhecimento e principalmente pelo compartilhamento do mesmo, sempre orientando e dando ideias para que pudéssemos produzir cada vez melhor, ou seja, um verdadeiro mentor.

Ao departamento de Engenharia Elétrica, do qual o LABSAC faz parte. Por proporcionar recursos e local para desenvolvimento tecnológico e intelectual.

Ao Departamento de Ciência da Computação, por todos esses anos dando suporte ao ensino de qualidade e pela grade de professores comprometidos com a formação de profissionais capazes para o mercado.

À Fundação de Amparo à Pesquisa e ao Desenvolvimento Científico e Tecnológico do Maranhão (FAPEMA) e ao Conselho de Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelo aceite e apoio aos projetos propostos do qual fiz parte e pelo contínuo incentivo à pesquisa científica a nível estadual e nacional.

À banca pela presteza em ter aceitado o convite, pela orientação, dicas e apontamentos que ajudarão a melhorar este trabalho.

Ao professor Mário Antonio Meireles Teixeira por ter aceitado o convite para a orientação com vistas à conclusão deste trabalho, dado os acontecimentos. Orientação esta que foi de estrema importância.

"Recentes experiências com ataques na Internet e principalmente o tremendo aumento na velocidade de propagação de ataques que se auto propagam claramente mostra que o problema de explorar vulnerabilidades de hosts não pode ser combatido apropriadamente com uma abordagem que apenas visa defender de ataques a sistemas finais consertando brechas na segurança quando os patches estiverem disponíveis".

Patrick Diebold (et al., p.1, 2005).

RESUMO

Esta monografia trata sobre o refatoramento do “HoneypotLabsac”, um *framework* para *Honeypot* em dispositivos móveis *Android*. Tem como objetivo de desenvolver um modelo de técnica de decepção que possa colaborar com o conhecimento sobre ataques cibernéticos no Sistema Operacional *Android*, e que possa contribuir com mecanismos de segurança já existentes, como IDS, IPS, WIDS e *Firewalls*. Utiliza-se de metodologia de decepção, na forma de *Honeypots*, um mecanismo que tem o intuito de estudar o comportamento dos atacantes por meio da criação de um ambiente próprio para receber ataques. A principal contribuição consiste na implementação de um protótipo de *software* intitulado “HoneypotLabsac 2.0”, visando aprimorar um projeto preexistente (*HoneypotLabsac*). Este trabalho almeja colaborar com os estudos de técnicas de segurança para dispositivos móveis *Android*.

Palavras-chave: Segurança de rede. Dispositivos móveis. Plataforma *Android. Honeypot*.

ABSTRACT

This work deals with the refactoring of "HoneypotLabsac", a framework for Honeypots on Android mobile devices. It aims to develop a deception technique template that can collaborate with knowledge about cyber-attacks on the Android operating system, and which can contribute to existing security mechanisms such as IDS, IPS, WIDS and Firewalls. Deception methodology is used in the form of Honeypots, a mechanism that aims to study the behavior of the attackers by creating an attack environment. The main contribution is the implementation of a software prototype entitled "HoneypotLabsac 2.0", aimed at improving a preexisting project (HoneypotLabsac). We aim to collaborate with studies of security techniques for Android mobile devices.

Palavras-chave: Network security. Mobile devices. Android platform. Honeypot.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS

IDS - Intrusion Detecting System
IPS - Intrusion Prevention System
WIDS - Wireless Intrusion Detection System
IDC - International Data Corporation
OHA - Open Handset Alliance
ASF - Apache Software Foundation
AOSP - Android Open Source Project
SO - Sistema Operacional
API - Application Program Interface
IPC - Inter Process Communication
CPU - Central Process Unit
ART - Android Runtime
GPS - Global Positioning System
SMS - Short Message Service
MMS - Multimedia Message System
VM - Virtual Machine
APK -Android Application Pack
GC - Garbage Collector
UID - User Identifier
OTA - Over The Air
C2DM - Cloud-to-Device-Messaging
CTS - Compatibility Test Suit
LOG - Registro de eventos
UXSS - Universal Cross-Site Scripting
IMSI - International Mobile Subscriber Identity
IMEI - International Mobile Equipment Identity
LBL - Berkeley Laboratory
DTK - Dalvik Toolkit
IP - Internet Protocol
SDK - Software Development Kit
IDE - Integrated Development Environment
ADB - Android Debug Bridge

SUMÁRIO

1 INTRODUÇÃO

Com o aumento da produção e do uso de dispositivos móveis observa-se o progresso do cenário global de compartilhamento e acesso à informação por meio dos dados interligados em rede. Dessa forma, tornou-se notável a otimização na resolução de problemas proporcionada pelas ferramentas que permitem a recuperação de dados online deixados pelo usuário. O citado "[...] sonho da 'informação na ponta dos dedos em qualquer lugar e a qualquer momento' e a visão lúdica e até certo ponto ficcional de décadas passadas foram capazes de antecipar muitos dos desafios que se apresentam hoje aos dispositivos móveis" (BRAGA et al., 2012, p. 3). Além disso Braga et al. (2012, p. 2) explicou em seu estudo que com tais avanços, "[...] o aumento de poder de computação, a grande conectividade e o grande aumento recente da variedade de serviços. e aplicativos disponíveis nos dispositivos móveis põem os *smartphones* e os *tablets* em evidência como alvos de ataques de risco elevado". Tais ataques são capazes de comprometer o usuário por meio da exposição de dados pessoais solicitados por sites e/ou aplicativos que podem permanecer ativos, disponíveis em diferentes 'repositórios' conectados entre si.

Existem diversas plataformas ou Sistemas Operacionais para dispositivos móveis, dentre elas: *Android*, *Bada*, *BlackBerry, IOS, Mobile Linux*, *Symbian, Windows Mobile* etc. O qual destacamos o sistema *Android* pela sua grande abrangência no mercado mundial. Em 2017, um relatório publicado pela IDC – *Internacional Data Corporation*, mostrou um alcance de domínio de 85% pelo *Android* no mercado de *smartphones* comparado aos 14,7% do *IOS*, 0,1% do *Windows Phone*, e 0,1% dos outros. O mesmo relatório indica que o domínio já foi maior em alguns meses de 2016, representando 87,6% (IDC: ANALYZE THE FUTURE, 2017). Assim, são claras a solidificação e a adesão no mercado, portanto essa plataforma se torna uma das mais visadas pelos atacantes que procuram constantemente novas vulnerabilidades de segurança.

Como o *Android* é uma plataforma aberta, os usuários estão mais sujeitos a baixar aplicações que utilizam recursos privados, roubam dados, ou mesmo inutilizam o dispositivo. Devido a esse cenário de risco, surge a necessidade de se trabalhar a segurança nesta plataforma, desenvolvendo ou melhorando técnicas e ferramentas, tais como *Botnet's* e *Honeypot*, que possibilitam aprender com ataques e assim agir ou montar um conjunto de regras para combater e atenuar os danos causados, por exemplo, por *malwares*. Assim contribuindo com a comunidade ao dar uma ferramenta a mais para que se possa aprender e se proteger de ataques nesta plataforma.

O acesso à internet nos dispositivos móveis é dado por dois modos, via rede *Wi-Fi* ou por uma rede móvel. O acesso via *Wi-Fi* é dado por um ponto de acesso e muito geralmente a rede já possui um Firewall; já a internet via rede móvel é fornecida e controlada por uma Operadora de Telefonia, o que faz cada uma possuir diferentes modos de operar. (BALMAS, 2013)

Para este trabalho, consideram-se apenas conexões com acesso à rede via *Wi-Fi*, pois já se conhece o modo de funcionamento destas.

1.1 Objetivos

1.1.1 Geral

O objetivo deste trabalho é desenvolver um modelo de técnica de decepção que possa colaborar com o conhecimento sobre ataques cibernéticos[1] no Sistema Operacional *Android*, e que possa contribuir com mecanismos de segurança já existentes, como IDS, IPS, WIDS e *Firewall*.

1.1.2 Específicos

- Compreender o *framework* de *Honeypots*;
- Aprimorar o *framework* já existente, direcionado para o *Android*;
- Adicionar melhorias ao código (deixando-o mais legível) e utilizar novas tecnologias;
- Produzir um *Honeypot* virtual de baixa interatividade utilizando o *framework* aprimorado;
- Promover a utilização deste tipo de ferramenta, pois o reuso que ela proporciona permite concentrar-se em partes específicas dos serviços mais vulneráveis;
- Contribuir para o aumento do nível de segurança para os dispositivos móveis, em particular para a plataforma *Android*, permitindo estudo sobre dados coletados;
- Construir um framework capaz de lidar com informações coletadas com rede *Wi-Fi* por meio do *Honeypot* criado.

O restante deste trabalho está assim organizado: o **Capítulo 2** trata da Fundamentação para a construção deste trabalho, embasando os conceitos e técnicas usadas. Já no **Capítulo 3**, tem-se um resumo do trabalho ao qual este trabalho se baseou, para que se possa fazer uma comparação. O **Capítulo 4** trata do trabalho em si, mostrando o desenvolvimento deste trabalho, demonstrando seus passos. Finalmente, o **Capítulo 5** apresenta as conclusões e trabalhos futuros.

[1] Este conceito será tratado posteriormente na **seção 2.1.**

2 FUNDAMENTAÇÃO TEÓRICA

O *Android* surgiu em 2003, com fundação da *Android Inc*., na Califórnia, com um grupo de empresários liderado por Andy Rubin, em que este a definiu na época como: “dispositivos móveis mais inteligentes e que estejam mais cientes das preferências e da localização do seu dono". Em 2005 a *Google* adquiriu o *Android Inc*., surgindo a *Google Mobile Division* e em 2007 grandes com companhias (fabricantes como a *Samsung*, *Sony*, operadoras como *Sprint Nextel*, fabricantes de *hardware* como a *Qualcomm*, e encabeçada pela *Google*) se reuniram e formaram um consórcio de tecnologia, que foi chamado OHA (OPEN HANDSET ALLIANCE, 2017). As empresas envolvidas investiram bastante recursos tecnológicos para que o projeto *Android* surgisse com poder competitivo no mercado, resultando em 2008 no lançamento do primeiro *smartphone* rodando o *Android*. (MEYER, 2017).

Foi projetado para ser *open source* (código de fonte aberto), sob licença ASF (*Apache Software Foundation*) (APACHE FOUNDATION, 2017), pois acreditava-se que o mercado necessitava de uma plataforma que seguisse essas diretrizes, desta forma permitindo que desenvolvedores pudessem anexar suas customizações. Contudo, para manter todas essas contribuições sem que levassem a atualizações inconsequentes, a *Android Open Source Project* (AOSP) criou o *Android Compatibility Program* (Programa de Compatibilidade *Android*). Assim sendo, todo aquele que deseja colaborar deve fazer parte deste projeto, que define todos os requisitos do que vem a ser compatível. Uma evolução constante é alcançada, pois permite que novos recursos surjam o tempo todo e que sejam mais rapidamente incluídos para uma próxima atualização. (ANDROID OPEN SOURCE PROJECT, 2017).

O *Android* é uma plataforma que entrega um conjunto completo de ferramentas para desenvolvimento em dispositivos móveis. Ele é formado por Sistema Operacional, *middleware* e algumas aplicações chave já implementadas. Foi construído em cima de um *Kernel* do *Linux* que foi adaptado para dispositivos móveis. Este *kernel* utiliza uma máquina virtual especialmente modificada para dispositivos móveis, chamada *Dalvik*, que otimiza o uso dos recursos da memória e *hardware*. (OPEN HANDSET ALLIANCE, 2017).

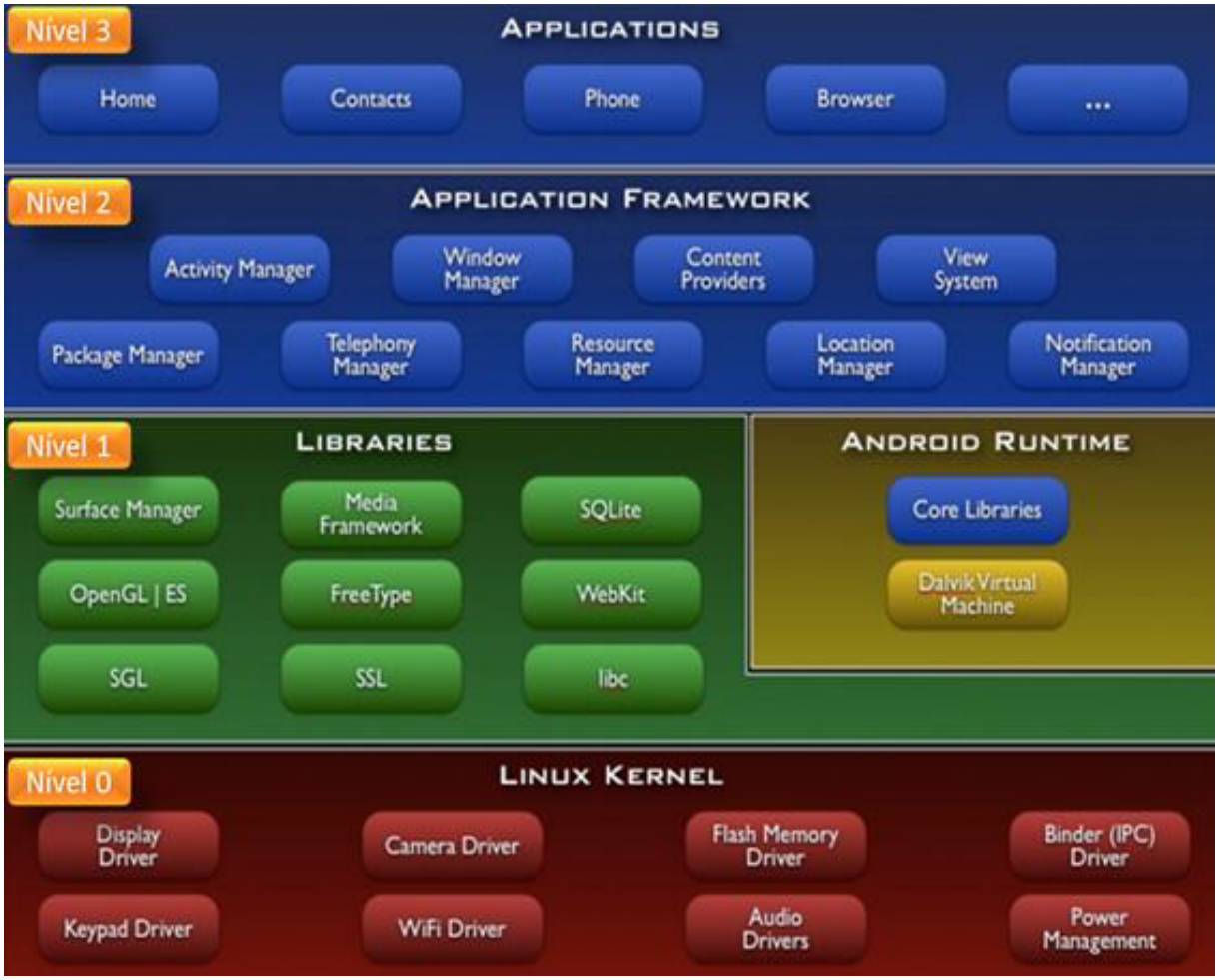
FIGURA 1 - Versões do *Kernel* e as respectivas versões do *Android*

Android Version	API Level	Linux Kernel Version
1.5 Cupcake	3	2.6.27
1.6 Donut	4	2.6.29
2.0/1 Eclair	5-7	2.6.29
2.2.x Froyo	8	2.6.32
2.3.x Gingerbread	9,10	2.6.35
3.x.x Honeycomb	11-13	2.6.36
4.0.x Ice Cream Sandwich	14,15	3.0.1
4.1.x Jelly Bean	16	3.0.31
4.2.x Jelly Bean	17	3.4.0
4.3 Jelly Bean	18	3.4.39
4.4 Kit Kat	19,20	3.10
5.x Lollipop	21,22	3.16.1
6.0.x Marshmallow	23	3.18.10
7.0.x Nougat	24	4.4.1
7.1.x Nougat	25	4.4.1

Fonte: Mi Community, 2017.

O SO *Android* atualmente está na versão 7 (Nougat) e sua base do *Linux Kernel* está na versão 4.4.1 (MI COMMUNITY, 2017). Na Figura 1 podemos ver a evolução das versões do *Android* e a correspondente adoção das versões do *kernel*. Incialmente foi baseado na versão 2.6. e após todas essas alterações passou a não ser mais uma distribuição *Linux* convencional.

FIGURA 2 – Camadas da arquitetura do Sistema Operacional *Android*



Fonte: (GOOGLE DEVELOPERS, 2017).

Na Figura 2 é retratada a arquitetura da plataforma *Android*, que é dividida em cinco setores, e o detalhamento de cada uma, segundo encontrado em (NOVICE'S BLOG, 2014) é mostrado a seguir:

Nível 0 (*Kernel):* Responsável pelo gerenciamento de memória, de processos e dos *drivers* do *hardware*, e pela segurança. O *kernel* também atua como uma camada de abstração entre o *hardware* e o *software*. O desenvolvedor não programa nessa camada pois os aplicativos criados fazem uso indireto do *kernel* através de APIs - *Application Programming Interface* (Interface de Programação de Aplicação) em camadas superiores. Também contém os *drivers* do sistema e com a customização para dispositivos móveis ganhou várias características específicas.

- **Binder**: é um mecanismo específico de IPC - *Inter Process Communication* (Comunicação entre processos) para *Android*, ou seja, gerencia a comunicação entre processos, responsável também pelo gerenciamento de memória e segurança;

□ **Wake Lock**: recurso que faz com que a CPU entre em modo de baixo consumo, o que garante economia de bateria;

□ **Power Management**: gerencia o uso da bateria, visando seu prolongamento. O sistema se utiliza de *wake locks* e mecanismo de tempo limite para trocar o estado de energia do sistema. A ideia é que os recursos de CPU sejam disponibilizados aos aplicativos por meio do *Android Framework*, tirando a CPU do estado de *wake lock*;

□ **Lower Memory Killer**: gerencia o uso da memória. Na prática os aplicativos geralmente nunca encerram, seu estado é salvado e o processo fica inativo. Ao se atingir um limite de uso, esse processo é chamado para assim encerrar processos de baixa prioridade;

□ **Ashman (Anonymous shared memory subsystem)**: é um sistema que permite que processos criem bloco de compartilhamento de memória nominal. Ou seja, pode ser usado para armazenar recursos que vários processos usam. Foi baseado no compartilhamento de memória do sistema *Linux* tradicional, permitindo ainda que o *kernel* possa requisitar esse espaço compartilhado assim que precisar de memória caso não esteja atualmente em uso.

Nível 1 (Bibliotecas): fornece componentes chaves do sistema, como a ART. Baseados em códigos que requerem bibliotecas nativas escritas em C e C++. Os desenvolvedores acessam as funcionalidades nativas através da API (*Application Program Interface)* do *framework* de aplicações[2], entre as mais importantes podemos citar:

- **SQLite**: para construção de armazenamento de dados estruturados, as transações armazenadas são mais leves;
- **Media framework**: permite reproduzir arquivos de áudio, vídeo e imagens. Suporta *plugins* de *codecs* de *hardware* e *software*;
- **Surface Manager**: permite compor as janelas da interface na tela e manipula toda a renderização para o *buffer* de quadros do dispositivo. Suporta renderização 2D e 3D.
    - **Nível 2 (Framework de aplicações)**: Fornece as APIs para desenvolvimento de aplicativos. Formado por diversos componentes essenciais que gerenciam as aplicações básicas do *smartphone*, sendo elas:

[2] Aplicações estas que são escritas em *Java*.

- **Activity Manager**: lida com o ciclo de vida da aplicação e provê um método de navegação em pilha;
- **Content Provider:** permite que aplicações possam ler e escrever dados (voluntariamente compartilhados) de outras aplicações e gerencia esse espaço compartilhado;
- **Location Manager:** permite acesso a localização por meio de GPS, torres de telefonia, entre outros;
- **Resource Manager**: gerencia vários tipos de recursos, como imagens, mídia, *strings* etc.

Nível 3 (Aplicações): É a camada superior da arquitetura *Android*. Fornece aplicações nativas para funcionalidades básicas do dispositivo. Algumas dessas funcionalidades são: aplicações de SMS (*Short Message Service*) e MMS (*Multimedia Message System*), configurações, discador, mapas, navegador, contatos, entre outros. Apesar de serem nativas, elas podem ser substituídas por aplicações de terceiros;

Android Runtime (ART): Este é o coração do *Android*; fornece a maior parte das funcionalidades disponíveis. A partir da versão 5.0 do *Android*, ela substitui a Máquina Virtual *Dalvík* (*Dalvik VM*), que é uma máquina virtual baseada em registradores, projetada pela *Google* especialmente para o *Android* (GOOGLE DEVELOPERS, 2017). A *Dalvík* foi projetada para dispositivos com baixo poder de processamento. Quando uma aplicação é compilada com a *Dalvik*, são gerados arquivos "*.class*", que depois são convertidos pela VM para "*.dex*" (*Dalvik Executable*), e por fim são compactados em um arquivo "*.apk*" (*Android Package File*). Toda aplicação executa em sua própria instância da *VM*. O *Android* usa o conceito de *Zygote*, que é um processo da VM que inicializa no boot do sistema. Ele é responsável por gerenciar as instâncias da VM para as aplicações. Assim, para manter a compatibilidade com versões anteriores do sistema, a ART usa como fonte de entrada o mesmo *bytecode*, por meio dos arquivos "*.dex*". De acordo com Ehringer (2010) suas principais características são:

□ **Ahead-of-time (AOT) compilation**: Antes do tempo de execução a ART compila um app com uma ferramenta denominada *dex2oat*. Com isso o processo de execução da aplicação fica mais rápido em relação ao *Dalvik* reduzindo o consumo de energia, e por consequência melhora a autonomia da bateria;

□ **Garbage Collector (GC) melhorado**: o GC pode impactar bastante na performance, reduzindo bastante o tempo de resposta das interfaces. As melhorias incluem: processamento paralelo durante as pausas do GC, coletor com tempo

reduzido para processos de vida curta alocados recentemente, compactação do GC para reduzir o uso e fragmentação de memória em segundo plano.

Com a Figura 2 também podemos abranger a questão da segurança que a plataforma *Android* provê. As camadas funcionam de uma forma hierárquica começando do topo à base, e cada camada mostrada assume que a camada logo abaixo está protegida, com exceção de uma pequena parte do SO, que executa com privilégio de *root* (administrador). E todo código que está acima da camada de *kernel* executa dentro do *Application Sandbox*.

O funcionamento do *Application Sandbox* ocorre da seguinte forma: a plataforma *Android* faz uso do gerenciamento de recursos do sistema *Linux*, que isola os aplicativos uns dos outros protegendo contra os que são maliciosos. Para alcançar esse isolamento cada aplicativo recebe um UID – *User Identifier* (Identificador de usuário) exclusivo e executa em seu próprio processo. Resumindo, o *Android* cria essa Caixa de Proteção (*Sandbox*), utilizando esse UID em nível de *kernel*. A *sandbox* é simples, auditável e baseada em décadas de melhoramentos do estilo UNIX de separação de processos e permissões de arquivos. (ANDROID OPEN SOURCE PROJECT, 2018b). Portanto, tal proteção é alcançada pois os processos não conseguem interagir entre si, a não ser que as devidas permissões sejam concedidas, e têm acesso limitado ao SO.

O *Android* foi projetado com segurança multicamada que é flexível o suficiente para suportar uma plataforma aberta enquanto protege os usuários da plataforma. O controle de segurança foi projetado para diminuir o peso sobre os desenvolvedores, pois desenvolvedores com experiência em segurança podem facilmente trabalhar com e confiar em controles de segurança flexíveis e os menos experientes serão protegidos por padrões seguros. Muitos são os modos que a plataforma *Android* provê uma plataforma estável para desenvolver, como por exemplo o serviço do *Play Services*, que entrega atualizações de segurança para bibliotecas de *software* críticas, como o *OpenSSL*, que é usada para proteger a comunicação entre aplicativos.

Os usuários olham as permissões requisitadas por cada aplicativo e têm controle sobre essas permissões. Essa abordagem tem a expectativa de que os atacantes tentariam realizar ataques comuns, como ataques de engenharia social para convencer o usuário instalar *malwares*, e ataques a aplicativos de terceiros (não nativos) (ANDROID OPEN SOURCE PROJECT, 2018a).

Dessa forma a segurança multicamada citada se dá com os três blocos principais da plataforma *Android*, denominadas Camadas de Segurança (ANDROID OPEN SOURCE PROJECT, 2018a), são elas: *Hardware*, Sistema Operacional e *Android Application Runtime*:

- **Hardware**: o *Android* é executado em diversos tipos de configuração de *hardware*, que incluem *smartphones*, *tablets*, *smart TV*, etc. O *Android* não conhece previamente o *hardware*, mas consegue tirar vantagem de algumas capacidades específicas que ele oferece, como o ARM *eXecutor-Never*[3]*;*
- **Sistema Operacional**: construído em cima do *kernel* do *Linux*, todos os recursos do dispositivo, como funções da câmera, dados de GPS, funções do *Bluetooth*, do telefone, conexões de rede, são acessados através do Sistema Operacional;
- **Android Application Runtime**: as aplicações *Android* são mais frequentemente escritas em *Java* e executam no *Android Runtime* (ART). No entanto, muitas aplicações incluindo serviços e aplicações raiz, são aplicações nativas ou usam bibliotecas nativas. Ambos os tipos, ART e nativos, executam no mesmo ambiente de segurança, o *Application Sandbox.*

A *Google* também provê serviços baseados em nuvem para promover segurança, mas são disponibilizados apenas para dispositivos compatíveis com dispositivos com o *Google Mobile Services*. E apesar destes serviços não pertencerem ao AOSP, eles estão disponíveis em muitos dispositivos *Android*. Os principais serviços disponibilizados são:

- **Google Play**: disponibilizada como uma coleção de serviços e aplicações que dão ao usuário a opção de descobrir e obter outros aplicativos para o seu dispositivo Ele permite avaliação dos aplicativos pelos usuários, verificação de licença de *software*, escaneamento de aplicação, etc;
- **Android updates**: esse serviço serve para receber atualizações de sistema. Atualizações essas que pretendem disponibilizar mais e melhores serviços de segurança para o dispositivo. Essas atualizações podem ser feitas por meio da WEB ou via OTA (*Over the Air*);
- **Application services**: disponibilizam *frameworks* que permitem que as aplicações ganhem capacidade de usar serviços da nuvem, como fazer *backup* dos dados e configurações do aplicativo e usar o serviço *cloud-to-device-messaging* (C2DM), para receber notificações.
- **Verify apps**: pode alertar ou até bloquear aplicativos prejudiciais, e continuamente faz escaneamento nas aplicações alertando ou removendo possíveis ameaças;
- **SafetyNet**: é um sistema de detecção de intrusos que ajuda a *Google* no rastreamento e restringir ameaças conhecidas, assim como detectar novas;

[3] Mais informação sobre essa tecnologia pode ser encontrada em (SINGH; VANEET, 2014)

- **SafetyTest Attestation**: é uma API de terceiros (que não é da *Google*) que determina se o dispositivo é compatível via CTS - *Compatibility Test Suit* (Conjunto de Testes de Compatibilidade). O resultado desse teste também pode ajudar a identificar a comunicação entre o aplicativo cliente e o servidor;
- **Android Device Manager**: serviço para ajudar na localização de dispositivos perdidos ou roubados.

2.1 Ataques cibernéticos: tipos e exemplos de *malware*

Conceitua-se ataque cibernético como: "[...] ciberataque [...] uma ação praticada por hackers que consiste na transmissão de vírus [...] que infectam, danificam e roubam informações de computadores e demais bancos de dados online, por exemplo". (SIGNIFICADOS, 2017). Qualquer software pode conter vulnerabilidades a ataques, permitindo que vírus ou *malwares*

> [...] invadam o sistema e comprometam sua integridade. Estas vulnerabilidades são descobertas pelo desenvolvedor do sistema, por usuários, ou, em um pior cenário, por criminosos. E, qualquer destes casos, o desenvolvedor precisa acelerar o processo para corrigi-las e disponibilizar uma atualização com a vulnerabilidades eliminada. (SCOTA et al., 2010, p. 1).

Existe quatro conceitos básicos a se considerar que ajudam a identificar e prevenir falhas na construção de softwares, são elas: confidencialidade, integridade, disponibilidade e legitimidade:

QUADRO 1 - Conceitos sobre segurança de dados

CONFIDENCIALIDADE	Garante que as informações armazenadas em um sistema de computação ou transmitidas através de uma rede de computadores, sejam acessadas ou manipuladas somente pelos usuários devidamente autorizados.
INTEGRIDADE	Garante que a informação processada ou transmitida, chegue ao seu destino exatamente da mesma forma como partiu da origem.
DISPONIBILIDADE	Garante que o sistema continue operando sem degradação de acesso e provê recursos aos usuários autorizados quando necessário. Diz respeito aos dados, aplicações, o dispositivo e a rede.
LEGITIMIDADE	Garante que os recursos não sejam utilizados por pessoas não autorizadas ou de forma não autorizada

Fonte: (SCOTA et al., 2010).

Afim de construir um software que esteja de acordo com esses conceitos, faz-se a correspondência com este trabalho:

A confidencialidade está presente no projeto pois os arquivos de *log* gerados são armazenados dentro da área restrita ao aplicativo, sendo que estes são compartilhados de tempo em tempo para um servidor de confiança para serem analisados.

A integridade está presente pois mesmo que o envio dos *logs* falhe num determinado momento, eles podem ser enviados novamente, por conta do funcionamento cíclico do compartilhamento.

A disponibilidade está parcialmente presente, pois a aplicação não é distribuída, portanto se a bateria do dispositivo acaba, por exemplo, as conexões não poderão occorrer. Porém os dados ficam salvos na memória interna e são mantidos enquanto os dados da aplicação não são forçadamente excluídos ou ela não é desinstalada.

A legitimidade é alcançada através da arquitetura da plataforma *Android*, pois ela esconde dentro de uma pasta na memória específica à aplicação criada, e apenas por meio dela é possível acessar.

Falando um pouco sobre as ameaças existentes à plataforma *Android*, destacamos os *malwares*, um dos maiores contaminadores do *Android*. Existem diversos tipos, mas segundo Jones (2017) os quatro tipos mais comuns de *malwares* nos dispositivos *Android* são:

- **Ransomware**: é um tipo de *malware* que transforma seu dispositivo num refém enquanto você paga ao cibercriminoso o seu resgate. Usualmente o pagamento é feito por *bitcoins*, ou outra forma não rastreável de transação monetária;
- **Universal Cross-Site Scripting *(*UXSS*)* Attack*:*** se você está usando um dispositivo *Android* antigo (anterior ao *KitKat* -versão 4.4), então seu dispositivo está suscetível a esse ataque. Ele é dado por um código em *JavaScript*, e se for clicado (enquanto está em seu navegador *web*), o *hacker* pode automaticamente baixar um aplicativo malicioso no seu dispositivo. A melhor forma de se livrar desse ataque é usando versões mais atuais do *Android*;
- **Malware escondido em aplicativos baixados:** alguns dos aplicativos mais inócuos possuem *malwares* escondidos. Até mesmo os jogos de paciência e aplicativos de histórico baixam vírus para os dispositivos. O problema com esses aplicativos é que eles podem estão funcionando bem por semanas ou mesmo meses antes de baixarem o vírus. E passado esse tempo, fazer a ligação desse aplicativo baixado como fonte dos seus novos vírus fica quase impensável;
- **Sequestro do instalador do *Android:*** um número muito grande de dispositivos – perto de 50% - estão em risco para esse vírus. O aplicativo do tipo *Android Installer Hijacking* (sequestro do instalador do *Android*) entra em ação quando você tenta baixar um aplicativo

válido. No entanto, o sequestrador instala um aplicativo malicioso. Enquanto as permissões estão sendo analisadas, o vírus sequestrador configura um aplicativo com aparência inocente que depois irá instalar o *malware*.

Pode-se encontrar uma listagem com os principais *malwares* da atualidade no site "Spreitzenbarth" (T-CONSULTING SPREITZENBARTH, 2016). Aqui citamos alguns exemplos, mostrando a diversidade de formas em que comprometem a segurança:

- **AccuTrack**: aplicação capaz de transformar o dispositivo em um rastreador GPS;
- **Ackposts**: este *trojan* rouba as informações de contato e os sobe para um servidor remoto;
- **Adsms**: este *trojan* tem permissão de envio de SMS. O canal de distribuição desse *malware* é por um SMS que contém o *link* para *download*;
- **BankBot**: tenta roubar as informações confidenciais e o dinheiro de aplicativos de banco;
- **Basebridge**: passa adiante informações confidenciais (SMS, IMSI, IMEI) para um servidor remoto;
- **CopyCat**: é um *Ad Network* agressivo que tem como principal objetivo gerar renda.

2.2 *Honeypots*: metodologia para iludir

A primeira aparição de mecanismos com intuito de iludir o atacante foi na década de 1980, quando em 1986 começaram ataques persistentes aos computadores do *Lawrence Berkeley Laboratory* (LBL); então Clifford Stoll criou um mecanismo de monitoramento (o que seria depois denominado *Honeypot*) para capturar o atacante e fazer relatório dos seus atos. (STOLL, 1988).

Já em 1990, a invasão monitorada foi a do laboratório da AT&T, em que Bill Cheswick conseguiu obter o detalhamento do ataque, que teve como principal objetivo explorar falhas do serviço *Sendmail*, que permitia obter acesso ao *gateway* do laboratório. Essa experiência permitiu aprender e a detectar as técnicas utilizadas pelo invasor. (CHESWICK, 1992).

Em 1998, foi feito o primeiro *Honeypot* com solução baseada em *software* de código aberto, que foi denominado DTK (*Deception Toolkit*), e foi desenvolvida por Fred Cohen. Esta ferramenta se baseou em emular vulnerabilidades do UNIX, para assim enganar atacantes e possivelmente aprender com as suas técnicas. (COHEN, 2006).

Em 1999, Lance Spitzner liderou um grupo de profissionais da segurança especializado em aprender as técnicas dos atacantes, e esse grupo formou o *Honeynet Project* ("*The Honeypot Project*", 2018). Foram realizados dois anos de pesquisa resultando no lançamento de um livro denominado "*Know Your Enemy*" que descrevia com detalhes as tecnologias de *Honeypot.*

Os *Honeypots* possuem meios para registrar as atividades dos atacantes. Podem ser bastante úteis no âmbito de segurança da rede, pois têm como objetivo principal enganar o invasor, fazendo-o pensar que está invadindo livremente um sistema operacional ou um serviço qualquer oferecido, porém, na verdade, está sendo constantemente monitorado e todos os seus passos registrados.

Segundo Hoepers et al. (2007) é um recurso de segurança: "[...] preparado para ser sondado, atacado ou até mesmo comprometido", que emula diversas vulnerabilidades e registra as informações sobre os ataques sofridos.

Uma forma comum de fazer esse registro é com arquivos de texto, chamados *logs*. Estes *logs* comumente contém os endereços IP, tanto o de origem (atacante) quanto do destino (atacado), as portas de conexão, um *timestamp* (registro do tempo), além do registro da atividade. (DIEBOLD et al., 2005)

Esse registro pode, por exemplo, ser um arquivo contendo informações como: data e hora do ataque, IP de origem (atacante) e de destino (atacado), tipo de ataque, etc. Esses arquivos primeiramente são armazenados localmente no dispositivo alvo, e posteriormente, esses *logs* podem ser enviados com intuito de armazenamento, segurança, e também análise para um IDS (*Intrusion Detection System*). Após feita a análise, caso seja detectada alguma ação suspeita, o IDS pode executar automaticamente contramedidas, ou apenas mandar um alerta. Este recurso é muito bem-vindo, principalmente para dispositivos móveis, que além de estarem expostos às vulnerabilidades de rede *wireless* são dispositivos com limitações físicas (bateria, memória, armazenamento interno). Assim sendo, o processamento de análise e armazenamento dos *logs* de ataques normalmente é deixado para o IDS. (SILVA, 2008)

Comumente é dito que existem dois tipos de *Honeypots,* o de Baixa interatividade e de Alta interatividade. Porém Spitzner (2002) faz referência também a *Honeypots* de média interatividade, situado entre a baixa e alta interatividade. No Quadro 2 tem-se uma comparação entre os níveis de interatividade possíveis em um *Honeypot*:

QUADRO 2 – Interatividade das características importantes

CARACTERÍSTICAS	NÍVEL DE INTERATIVIDADE		
	BAIXO	MÉDIO	ALTO
Trabalho para instalar e configurar	Fácil	Mediano	Difícil
Trabalho para ativar e manter	Fácil	Mediano	Difícil
Coleta de informações	Limitada	Variável	Extensa
Nível de risco	Baixo	Médio	Alto

Fonte: Spitzner, 2002.

- **Baixa Interatividade**: comumente simulam serviços, ou seja, emulam os componentes pretendidos a ser atacados, como desde uma porta até um serviço de rede completa. Geralmente funcionam por meio de *scripts* simples para simular a interação com o atacante. São fáceis de configurar e se tornam até certo ponto seguros por terem uma capacidade baixa de interações. Foram estruturados para capturar comportamentos já conhecidos, ou seja, o atacante age de uma forma já determinada. Um ponto negativo é que é bem mais fácil para o atacante descobrir que se ele está interagindo com um *Honeypot*. (DIEBOLD et al., 2005)

- **Alta Interatividade**: os atacantes interagem diretamente com aplicações e serviços reais, porém com em um ambiente controlado. Este tipo oferece uma maior interação com o atacante, visto que o acesso acontece ~~a~~ em um dispositivo real com todas suas características, mas teoricamente ele não terá acesso ao restante do sistema. Um ponto positivo é que neste modelo o atacante será quase incapaz de descobrir que ele está interagindo com um *Honeypot* (DIEBOLD et al., 2005).

E finalmente o tipo não muito comum, o qual é citado pelo autor Spitzner (2002):

- **Média Interatividade**: esta categoria é um pouco mais complexa do que a de baixa interatividade, os tornando mais difíceis de instalar e configurar. Necessita de um alto grau de implementação e customização, pois vai além de apenas escutar portas, e sim emula grande parte do sistema, como por exemplo um servidor *web* completo. Conseguem com isso uma maior captura de informações dos atacantes, chegando até a capturar as ferramentas que usam.

2.3 Frameworks

O conceito básico/geral de *framework*, segundo o dicionário Merriam-Webster (“Definition of Framework”, [20-?]) é uma estrutura conceitual básica; um esqueleto ou moldura estruturada.

A definição de Johnson (1997, p. 39) é: “um *framework* é um padrão reusável de toda ou parte de um sistema que é representado por um conjunto de classes abstratas e como suas instâncias interagem”, ou seja, é uma técnica para criação de um modelo para desenvolvimento de um sistema. Um conjunto de componentes relacionados entre si com um objetivo específico.

Frameworks são uma forma de reuso de modelo ou arquétipo. São similares à outras técnicas de reuso de alto grau como *Templates* (LI et al., 1992) ou *Schemas* (GERO, 1990). A principal diferença é que *frameworks* são expressos com uma linguagem de programação. Fato é que esse padrão torna a vida do desenvolvedor mais fácil, pois são fáceis de aprender e usar, são reusáveis, gerando aplicações semicompletas para assim criar sistemas especializados.

Neste trabalho iremos usar o *Framework* de Aplicação Orientado a Objetos. Que segundo Fayad e Schimidt (1997) “são uma tecnologia promissora para objetificar designs e implementações de software já validados, para reduzir custos e aumentar a qualidade do código”. Podem ser classificados quanto à forma usada para estendê-los:

- **Tipo caixa-branca (*white box*)**: têm uma ligação forte com as funções de linguagem OO (orientadas a objeto), como herança e acoplamento fácil, que promovem fácil extensão. Pode ser usado das seguintes formas: permite (a) herança das classes do *framework* e (b) sobrescrição de métodos chave (pode ser usado padrões como o *Template Method* (GAMMA et al., 1996)). Ele exige que o desenvolvedor que irá usá-lo o conheça bem sua estrutura. Tendem a produzir aplicações que têm forte dependência com partes específicas da hierarquia de classes do *framework*.

- **Tipo caixa-preta (*black box*)**: permite extensão (a) definindo classes que permitam criar interfaces para os componentes, criando instâncias através de configurações e composições ou (b) integrando estes componentes em um *framework* usando padrões como *Strategy* e *Functor* (GAMMA et al., 1996). São geralmente mais fáceis de usar e estender, mas são mais difíceis de implementar, pois deve fornecer funcionalidades que abrangem uma grande variedade de possíveis casos de uso.

Fayad, Schimidt (1997) também menciona a classificação pela **estrutura**:

- **System infrastructure framework:** voltado para desenvolvimento de infraestrutura de sistemas portáteis e eficientes como SO’s. são prioritariamente usados internamente como *software* da empresa e não é vendido ao usuário diretamente.

- **Middleware integration frameworks:** comumente usados para integrar aplicações distribuídas e componentes. São projetados para facilitar a modularização, reuso e extensão da infraestrutura do *software*, facilitando o trabalho em um ambiente distribuído.
- **Enterprise application frameworks (Frameworks de Aplicação):** servem para integrar aplicações e componentes distribuídos. Possibilitam trabalhar em um ambiente distribuído como se não o fosse, pois escondem o baixo nível de comunicação entre os componentes distribuídos. Constroem uma aplicação final.

Vantagens de se usar *frameworks*, segundo (LANDIN et al., 1995)**:**

- Reduz o tempo para o mercado: quando se cria aplicações em série que usam como base *frameworks*, apenas as funcionalidades diferentes de cada aplicação precisam ter seu código reescrito. Isso reduz o tempo de desenvolvimento, por consequência o tempo para ser disponibilizado para o mercado;
- Manutenção: gasta-se por volta de 60-85% do custo para manter uma aplicação de larga escala com constante manutenção. Ao se usar *frameworks* como base para criar aplicações apenas será necessário implementar um ponto específico, gerando mais precisão nas correções, reduzindo o volume de código retaforado;
- Testes: quando se reusa *frameworks*, os testes referentes também serão reusados. Os únicos testes que terão que ser implementados são os dos pontos que diferem entre as aplicações e os novos módulos criados;
- Confiabilidade: um *framework* pode conter, assim como demais aplicações, erros e *bugs*. Porém, a medida que ele é reusado, ele tende a se tornar estável e caso surja erros, eles serão percebidos e corrigidos mais rapidamente. Além do fato de que reusar um código estável aumenta a confiabilidade da aplicação final;
- Padrão: um *framework* que é bem estruturado e segue as 'normas' da empresa gera um conforto para os desenvolvedores implementarem novas aplicações;
- Incorporam expertise: desenvolver bons padrões de desenvolvimento demanda experiência. Ao utilizar *frameworks*, seus padrões e regras são utilizadas constantemente e, portanto, os desenvolvedores se preocupam em desenvolver as soluções para uma aplicação e utilizar as ferramentas bem fundamentadas fornecidas pelo *framework*;
- Promovem consistência e compatibilidade: há uma facilidade em se trabalhar com várias aplicações que compartilham da mesma estrutura. Conseguem ter uma boa integração no ponto de vista do usuário, pois têm uma interface similar ou igual.

3 PROJETO ANTERIOR: *HoneypotLabsac*

Neste capítulo será comentado de uma maneira resumida o trabalho de Oliveira (2012), chamado *HoneypotLabsac*, no qual nos baseamos para fazer a refatoração.

3.1 Framework 1.0: aplicabilidade

No trabalho de Oliveira (2012) é explanado que o *Framework* escolhido para o estudo foi o de classificação "Framework de Aplicação" (FAYAD; SCHMIDT, 1997) e gerou o "FrameworkLabsac". O diferencial deste *framework* é uma simplicidade em coletar dados de dispositivos *Android* conectados em redes *wireless* por meio do *Honeypot* virtual em nível de aplicativo.

3.2 Framework 1.0: arquitetura

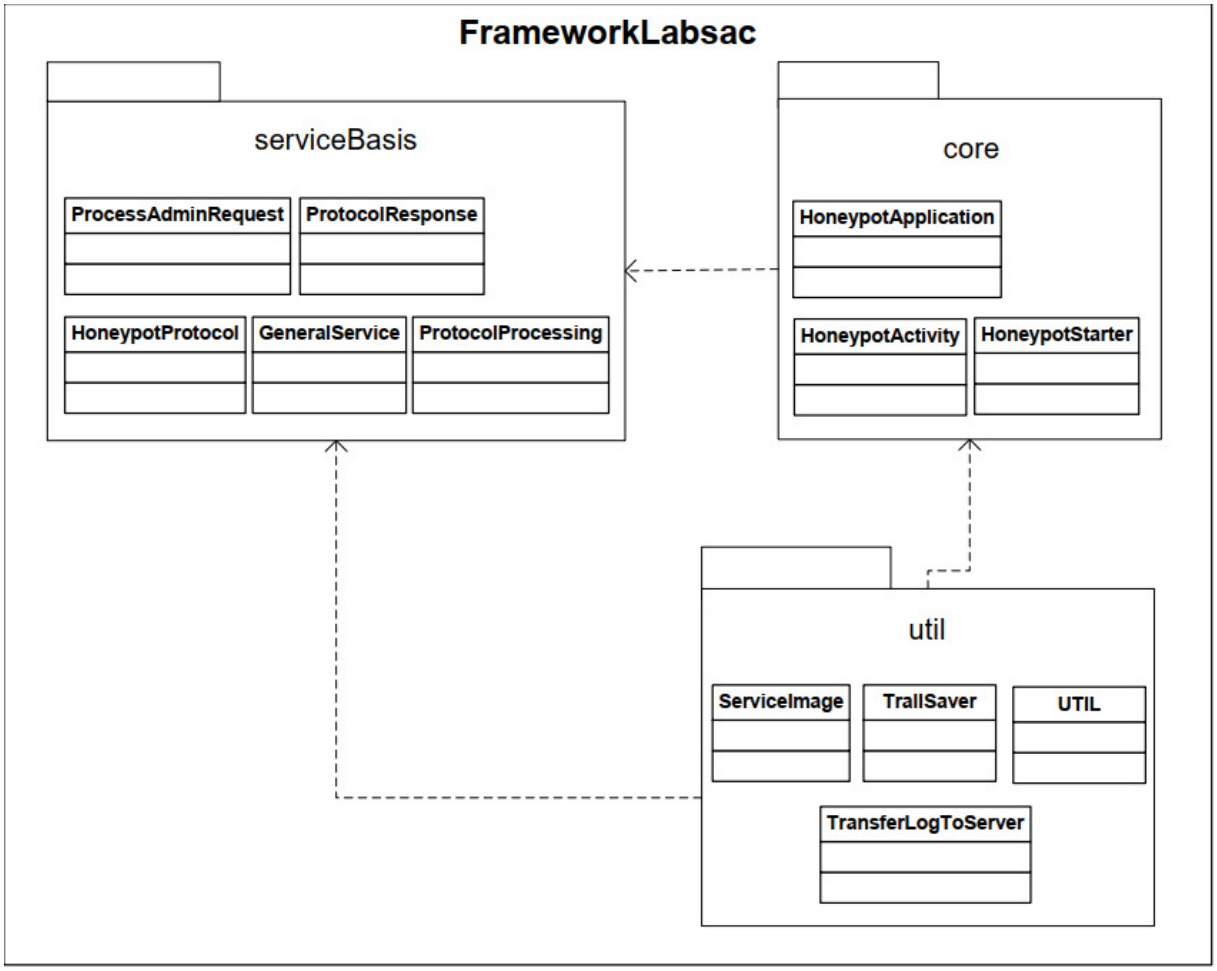
A finalidade do *FrameworkLabsac* é fornecer serviços a serem executados em *background*, não sendo estritamente necessária uma interface gráfica. Esse fator é uma vantagem para serviços de tempo indeterminado. Além disso, tem uma arquitetura simples, composta pelos seguintes componentes: *Kernel Linux*, SO *Android* e o *FrameworkLabsac*, como podemos ver na Figura 3:

FIGURA 3 - Arquitetura do *Framework 1.0*

Fonte: Oliveira, 2012.

A seguir veremos na Figura 4 a descrição visual da estrutura dos pacotes juntamente às classes que cada um contém:

FIGURA 4 - Diagrama de Pacote do *Framework 1.0*



Fonte: Oliveira, 2012.

3.3 Framework 1.0: utilização

De acordo com Oliveira (2012), o *FrameworkLabsac 1.0* utiliza linguagem de programação Java (ORACLE, [20-]) juntamente ao SDK *Android*[4], desenvolvido com a Eclipse (ECLIPSE FOUNDATION, 2018). O SDK *Android* é composto por: Emulador (QEMU), ferramentas utilitárias e API completa para Java. O *FrameworkLabsac* é formado por 12 classes que servem de base para a criação dos serviços da aplicação. São elas: *HoneypotActivity*, *HoneypotApplication*, *HoneypotStarter*, *GeneralService*, *HoneypotProcotol*, *ProcessAdminRequests, ProtocolProcessing, ProtocolResponse, ServiceImage, TrailServer, TransferLogToServer e UTIL.*

[4] Hoje esta tecnologia não está mais disponível desta maneira, sendo que o Ambiente de Desenvolvimento já traz todas as ferramentas necessárias.

Segundo Oliveira (2012), após análise comparativa entre *Honeydroid*, *HoneyM*, *Honeyd* e a proposta do *HoneypotLabsac*, foi possível constatar que o projeto proposto foi o primeiro a funcionar em nível de aplicação, como é descrito na tabela a seguir:

QUADRO 3 - Comparação qualitativa das características

Características	HoneyDroid	HoneyM	Honeyd	HoneypotLabsac
Nível de Aplicação	Não	Não	Não	Sim
Nível de Sistema Operacional	Não	Não	Sim	Não
Nível de Kernel	Sim	Não	Não	Não
Virtualização de Hardware	Sim	Não	Não	Não
Baixa Interatividade	Não	Sim	Sim	Sim
Sistema Operacional *Android*	Sim	Não	Não	Sim
Sistema Operacional Móvel	Sim	Sim	Não	Sim

Fonte: Oliveira, 2012.

3.4 HoneypotLabsac: aplicabilidade

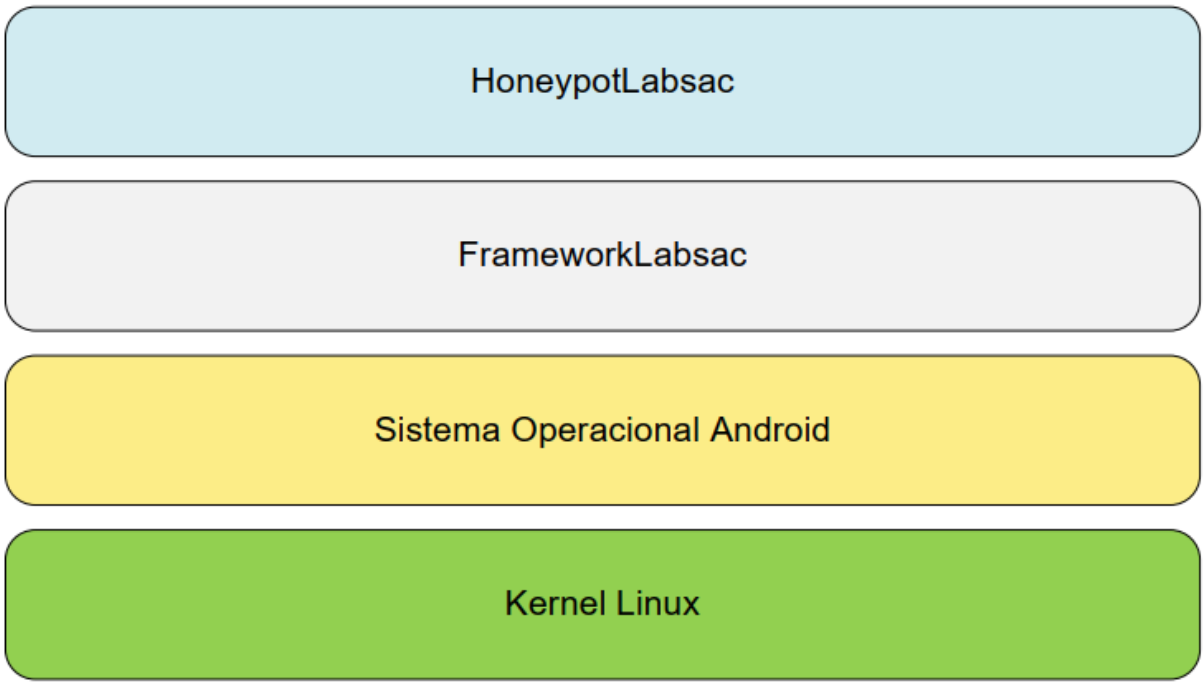
Neste trabalho ainda foi adicionado a classificação de *Honeypots* monitores de portas de comunicação, que escutam essas portas e geram *logs* de toda comunicação feita.

Também foi dito que a principal função de um *Honeypot* de baixa interatividade é "coletar especificamente tentativas de conexão e sondagens não autorizadas, além de alertar sobre ataques automáticos (*worms*), dentre outras pragas" (OLIVEIRA, 2012, p. 60). E que fornecem *logs* que contém as seguintes informações: data e horário do ataque; endereço IP e porta de origem; endereço IP e porta de destino; atividade do atacante.

3.5 HoneypotLabsac: arquitetura

A arquitetura da aplicação final é basicamente a mesma do *framework* apresentada na **seção 3.2**, como podemos ver na Figura 5 a seguir:

FIGURA 5 - Arquitetura do *HoneypotLabsac*



Fonte: Oliveira, 2012.

4 REFATORAÇÃO: HoneypotLabasc 2.0

Neste capítulo é descrito o produto deste trabalho, que consiste na construção de um *framework* de *Honeypot* virtual para dispositivos móveis *Android*, denominado *FrameworkLabsac* 2.0, sendo uma refatoração, e também um melhoramento do projeto de Oliveira (2012), abordado no **Capítulo 3,** e posteriormente a construção de uma aplicação denominada *Honeypot Labsac* 2.0. Será mostrado em cada seção a explicação do trabalho retaforado. Primeiramente será descrito o *framework*, e posteriormente o seu uso para a construção da aplicação de *Honeypot* mencionada.

4.1 Framework 2.0: aplicabilidade

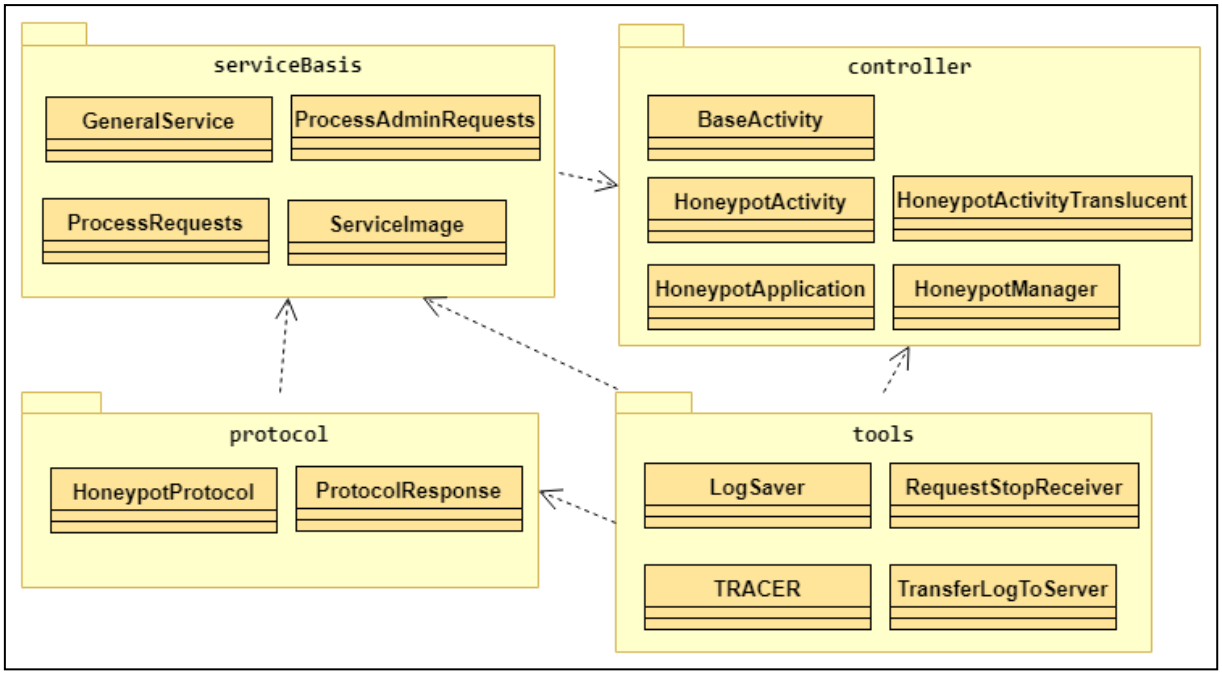
Este trabalho mantém a classificação quanto à estrutura como *Enterprise Application Framework* (Framework de Aplicação) (FAYAD; SCHMIDT, 1997), quanto à forma como Caixa-branca, pois exige que o desenvolvedor conheça a estrutura interna para construir cada parte do *Honeypot* e continua baseada na simplicidade de coletar dados de um dispositivo conectado a uma rede *Wi-fi*.

4.2 Framework 2.0: arquitetura

A arquitetura geral do *Framework* 2.0 continua a mesma, baseada no projeto anterior, como é mostrado na **seção 3.2** na Figura 3.

Já na arquitetura específica foram adotadas algumas modificações para uma melhor organização, tornando mais intuitivo o uso das funcionalidades oferecidas, renomeando algumas classes e pacotes, remanejando outras para outro pacote, criando classes. Esta arquitetura está representada na Figura 6 mostrada a seguir, seguida com a descrição dos pacotes. A descrição de cada classe dos pacotes será feita na seção posterior **4.3**.

FIGURA 6 - Diagrama de Pacote do *Framework 2.0*



Fonte: O Autor.

- Pacote **controller**: responsável pela criação do controle geral da aplicação final do *Honeypot*. Inicializa e gerencia os serviços, criando suas imagens (contendo as portas de comunicação e outras informações), verificando seu estado ativo. Verifica o estado da rede e fornece recursos para inicializar os componentes de tela quando forem necessários. É composto por cinco classes: *BaseActivity*, *HoneypotActivity*, *HoneypotActivityTranslucent*, *HoneypotApplication*, *HoneypotManager*;
- Pacote **protocol**: responsável por criar os componentes internos de comunicação, fazendo o direcionamento e verificação das trocas de mensagens e do controle de tráfego, bem como criar um modelo para responder as requisições feitas pelos clientes (atacantes) conectados. Constituído pelas classes: *HoneypotProtocol* e *ProtocolResponse*;
- Pacote **serviceBasis**: responsável por criar os componentes de comunicação. É a base para criação dos serviços (utilizando os *Services* do *Android*), fornecendo métodos para notificação, criando um modelo para interação com o serviço, etc. Constituído pelas classes: *GeneralService*, *ProcessAdminRequests*, *ProcessRequests* e *ServiceImage*;
- Pacote **tools**: fornece componentes auxiliares, como uma estrutura para passar informações de configuração aos serviços, suporte ao salvamento de *logs* na memória interna do dispositivo móvel para posteriormente mandá-los para um servidor, e contém uma estrutura para gerenciar mensagens de depuração.

Composto por cinco classes: *LogSaver*, *RequestStopReceiver*, *TRACER* e *TransferLogToServer*.

4.3 Framework 2.0: utilização

Aproveitando o quadro comparativo do trabalho anterior mostrado na **seção 3.3**, obtém-se o Quadro 4, e que mostra que as características que destacaram o trabalho anterior permanecem:

QUADRO 4 - Comparação qualitativa entre os projetos

Características	HoneypotLabsac	HoneypotLabsac 2.0
Nível de Aplicação	Sim	Sim
Nível de Sistema Operacional	Não	Não
Nível de Kernel	Não	Não
Virtualização de Hardware	Não	Não
Baixa Interatividade	Sim	Sim
Sistema Operacional *Android*	Sim	Sim
Sistema Operacional Móvel	Sim	Sim

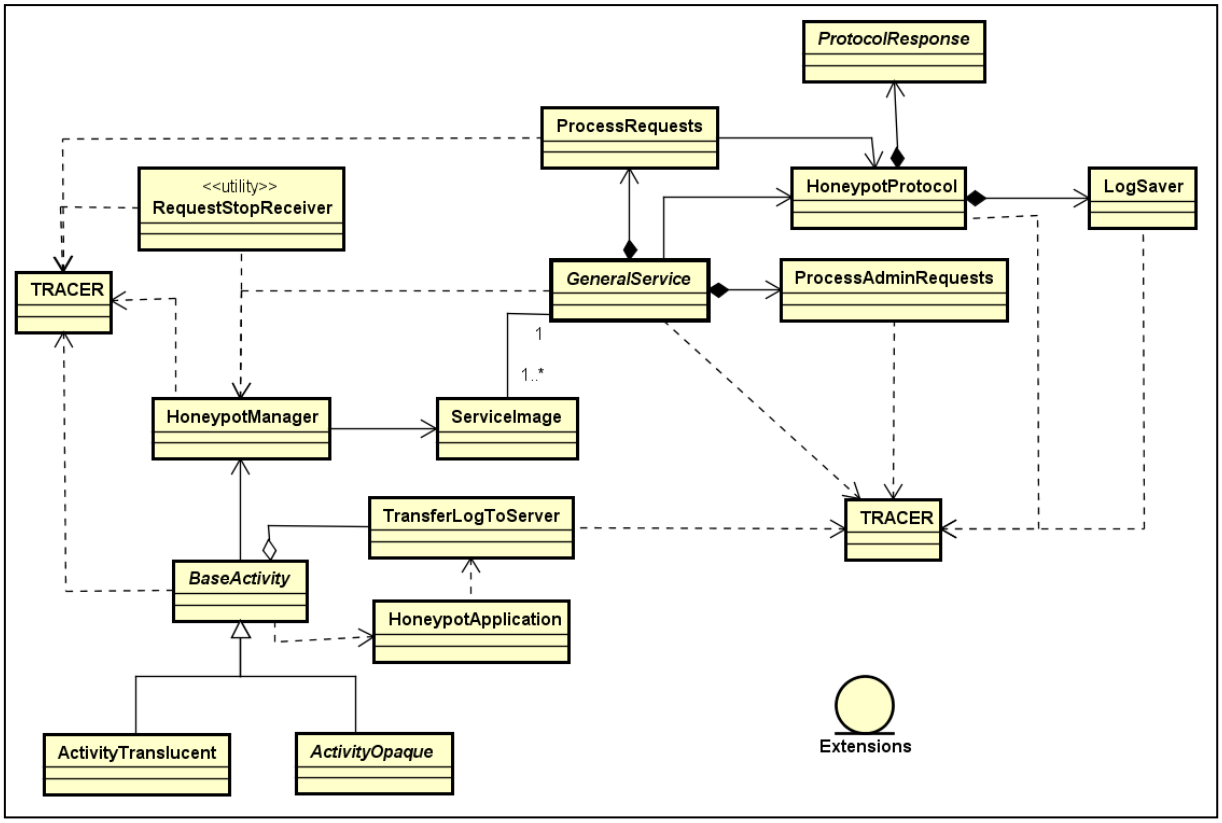
Fonte: O autor.

O *Framework Labsac* 2.0 foi implementado utilizando a linguagem *Kotlin* (ANDROID DEVELOPERS, [2016?]), segundo vemos em Heiss (2013) “o principal objetivo do Projeto *Kotlin* é criar para os desenvolvedores uma linguagem de propósito geral que pode servir como uma ferramenta útil que e segura, concisa, flexível e completamente compatível com o Java. Ela vem se tornando muito popular, tornando-se a 2ª linguagem ‘mais amada’ e a 4ª mais desejada/buscada, segundo uma pesquisa feita pelo *Stack Overflow* (STACK OVERFLOW, 2018). O anúncio da adoção oficial no *Android* foi feito no Google IO 2017 (SHAFIROV, 2017).

A IDE utilizada foi o *Android Studio* (GOOGLE DEVELOPERS, 2018a), da *JetBrains* (JETBRAINS, 2018), mesma empresa criadora da linguagem *Kotlin*. Ao utilizar o *Android Studio*, tornou-se possível disponibilizar o *Framework* 2.0 como um módulo, assim podendo ser facilmente incorporado em outro projeto, que o usará para criar o *Honeypot*.

O fato de ser um *framework* possibilita a sua fácil extensão, permitindo adicionar novos serviços de maneira ágil. O desenvolvedor que queira criar um serviço deve conhecer as 15 classes componentes. podemos ver o diagrama de classes na Figura 7, que serão descritas a seguir:

FIGURA 7 - Diagrama de Classe do *Framework* 2.0



Fonte: O autor.

- **BaseActivity:** classe abstrata, serve como base para a estrutura de *gerenciamento* da aplicação de *Honeypot*. Possui os seguintes métodos:

  - *startComponents():* método abstrato. Deve ser implementado na classe final do Honeypot. Contendo as inicializações e configurações de dados para os serviços e a aplicação no geral, assim como inicializar os componentes visuais quando assim for necessário;

  - *startServiceImages()*: método abstrato. Deve ser implementado para conter a inicialização das imagens de cada serviço;

  - *updateServices()*: método abstrato. É implementado para atualizar variáveis relacionadas aos serviços, verificando seus estados ativos;

  - *start()*: inicializa um serviço. Recebe a imagem de um serviço;

  - *stop()*: finaliza um serviço. Recebe a imagem de um serviço.

- ***ActivityOpaque*:** esta classe herda de *BaseActivity*. É abstrata, e caso o desenvolvedor queira criar uma interface de usuário ela deve ser escolhida como classe pai da *Activity* principal. Possui os seguintes métodos:

  - *scheduleAlarm()*: responsável por construir o disparador cíclico de envio dos *logs* ao servidor, por meio da classe *TransferLogToServer*;

- *updateLayout(time = 500)*: método para atualização dos componentes de layout após mudanças de início e término de serviços. Possui um tempo de "*delay*" padrão de 500 milissegundos;
- *rebootAlarm()*: caso haja mudança no tempo do ciclo, como por exemplo o usuário alterar o valor numa tela de configuração, esta função deve ser chamada.

- ***ActivityTranslucent*:** esta classe herda de *BaseActivity*. É abstrata, e caso o desenvolvedor não queira criar uma interface para a *Activity* principal ela deve ser escolhida, assim criando uma *Activity* "fantasma". Possui os seguintes métodos:
  - *startImages()*: serve para que o desenvolvedor implemente com a inicialização para cada serviço criado.
  - *toaster(args: String):* para chamar um *Toast* que avisa que determinado serviço está executando. A principal função é a de fornecer um aviso visual para o usuário que a aplicação está executando, já que a aplicação não possui interface. Pode receber várias *strings* com os nomes de cada serviço inicializado.
- ***HoneypotApplication*:** seu principal intuito é disponibilizar ferramentas globais à aplicação. Inicializa o componente *SharedPreferences*, e possui métodos de acesso essenciais à aplicação:
  - *getIntPreference()*: para recuperar dados do tipo *Int*;
  - *getLongPreference()*: para recuperar dados do tipo *Long*;
  - *getStringPreference()*: para recuperar dados do tipo *String*;
  - Todos esses métodos recuperam do *SharedPreferences,* e recebem um valor padrão, para ser usado caso o recurso pretendido não exista.
- ***HoneypotManager*:** dá suporte ao gerenciamento dos serviços, inicializando um serviço com sua porta e protocolo correspondentes. Para cumprir seu desígnio ela precisa do suporte da classe *ServiceImage*, do pacote ***serviceBasis***.
  - *checkNetwork(): verifica se existe conexão de rede;*
  - *checkPort()*: verifica se o serviço está executando. No caso verifica especificamente a porta administradora do serviço;
  - *startService()*: inicializa o serviço em si, passando os parâmetros necessários.
  - *requestStop()*: requisita a interrupção de um serviço.
- ***GeneralService*:** É uma classe abstrata de serviço do SO *Android* (tipo *Service*), ou seja, é a base para criação dos serviços específicos em que são executadas as tarefas em *background*. Seus métodos são:

- *onStartCommand()*: principal método, que deve ser obrigatoriamente sobrescrito na classe de um serviço. Aqui a notificação e os procedimentos de comunicação do serviço são inicializados;
- *parseExtras()*: recupera dados referentes à execução do serviço passados por *Intent*. Retorna *true* (verdadeiro) caso não ocorra erro;
- *stopEverything()*: responsável por interromper todos os processos referentes a um serviço;
- *notification()*: responsável por configurar a notificação, que será disparada quando um serviço estiver em execução;
- *createNotificationChannel()*: para APIs do *Android* a partir da versão 26. Serve para criar um centro de gerenciamento de notificação, chamado canal de notificação.

- ***ProcessAdminRequests:*** Classe utilitária que executa um procedimento cíclico que consiste em receber requisições na porta administradora de um serviço, ou seja, verifica o status do serviço. Esse procedimento fica executando em paralelo ao serviço correspondente.

- ***ProcessRequests*:** Classe utilitária que executa um procedimento cíclico de receber requisições dos usuários (atacantes), cria a estrutura de protocolo (*HoneypotProtocol*) com sua respectiva classe de resposta (*ProtocolResponse*).

- ***ServiceImage*:** Cria a imagem de um serviço, contendo as informações relativas a um serviço. Uma classe auxiliar, que é utilizada por outras classes, como a *HoneypotActivity.*

- ***HoneypotProtocol*:** Tem o objetivo de criar o Protocolo do serviço. Possui métodos que auxiliam a construção do canal de comunicação do atacante com o sistema e depois com a criação do *log*;

- ***ProtocolResponse*:** Classe abstrata. Dá suporte ao gerenciamento da comunicação dos serviços. Base para criação das respostas ao cliente de conexão (atacante);
  - *respond()*: método abstrato, chamado quando necessita-se responder à requisição do cliente (atacante). Deve ser implementado pelo desenvolvedor para conter as respostas de acordo com as requisições recebidas;
  - *flush()*: Todas as mensagens de resposta são acumuladas em um *streaming* de saída, e após a resposta estar pronta este método a despacha para o cliente.

- ***LogSaver*:** Responsável por criar e salvar os arquivos de *log*. Ficam salvos na memória interna, dentro do conteúdo privado do aplicativo.

- ***RequestStopReceiver*:** Classe que implementa um *BroadcastReceiver* responsável por receber a requisição de encerramento de um determinado serviço que é enviada exclusivamente através do botão de ação da notificação do mesmo.
- ***TRACER*:** Gerencia mensagens de depuração, ou seja, para o *Logcat*[5] da *IDE*.
- ***TransferLogToServer:*** Pega os arquivos de *log* gerados e os envia para um servidor. Os dados do servidor como IP e porta de comunicação são inicializados na inicialização da aplicação.

4.4 Framework 2.0: criando um serviço

A primeira diferença para o trabalho anterior é o uso da linguagem *Kotlin*. Ela nos permite escrever um código bem mais limpo, e em muitos casos com bem menos linhas de código. Permite, por exemplo, que o desenvolvedor ao utilizar o *Framework* 2.0 possa criar funções extras para ele, sem alterar o código fonte através do recurso chamado função de extensão. Permite evitar o *overhead* na alocação de memória ~~ao~~ na ~~chamar~~ chamada de funções utilizando a *inline functions*, e que funções (ou métodos) passem funções como parâmetro ou as usem como retorno, isso se chama funções de alta ordem, o que dá uma flexibilidade muito maior do que o código *Java* para o código[6].

No trabalho anterior a utilização do *framework* contém alguns pontos da estrutura que são problemáticos e fazem com que o mesmo seja parcialmente dependente. Estes pontos estão listados a seguir, e detalhado em duas partes, a primeira descrevendo como era feito no trabalho anterior, e depois como está atualmente, ou seja, mostra como melhorou:

- Criação do *ServiceImage*:

  Antes: Depende de um objeto *ENUM* chamado *RESPTYPE*, que tem por função listar os serviços existentes na implementação e dizem o tipo que está sendo utilizado na hora. Porém esses serviços não têm como serem conhecidos previamente pelo *framework*, o que faz com que esse *ENUM* seja implementado na aplicação estendida, descaracterizando sua função de *framework*.

[5] Ferramenta para exibir mensagens de sistema, como ocorrências do Garbage Collector, oumensgens enviadas pela classe Log. Para mais informações veja em: <https://developer.android.com/studio/debug/am-logcat>. Acesso em 15 de set. de 2018

[6] Outras caraterísticas podem ser encontradas no documento intitulado “Kotlin Language Documentation”. Disponível em: <https://kotlinlang.org/docs/kotlin-docs.pdf>. Acesso em: 08 set. 2018.

Depois: Não depende mais de informação externa, essa classe apenas tem a função de encapsular (guardar tornando um objeto isolado) as informações que o serviço necessitará (portas, classe do serviço, nome para o serviço).

- Atribuindo *Responses* para *HoneypotProtocol*:

  Antes: Novamente depende do *ENUM* chamado *RESPTYPE*, para que dentro do *HoneypotProtocol* a instância correspondente do *HoneypotResponse* seja resgatada.

  Depois: Para acabar com a dependência para resgatar o *Response* correto, na criação do *Service*, ele recebe uma instância de *HoneypotProtocol*, e este recebe uma instância do *HoneypotResponse* correspondente.

- Na criação do Service específico:

  Antes: Além de passar os dados obrigatórios para criar a notificação de serviço (nome do serviço, id, ícone) também existe um dado opcional que é o texto de "chamada" que aparece logo que a notificação aparece, ela também é instanciada aqui, gerando o inconveniente de ter que se lembrar disso.

  Depois: Obrigatoriamente só precisa passar para a *Generalservice* a instância do *HoneypotProtocol*, e os dados para a notificação de serviço, tanto os obrigatórios quando o opcional. Mas a instância do *Protocol* é requerida no construtor da classe, portanto não há como esquecer de inicializá-lo.

- Salvamento dos arquivos de *log*:

  Antes: Depende primeiramente que os nomes dos arquivos sejam pré-carregados por meio do *Resource String*[7] e sejam associados a cada *RESPTYPE* (um Enum que relaciona o tipo de resposta para cada serviço existente) correspondente através da modelagem de dados chave-valor. Essa modelagem de dados é acessada através da classe *ExecutorResponse*.

  Depois: O nome do arquivo de *log* é derivado do "nome do serviço" passado ao iniciar sua execução. Não necessita de qualquer tratamento do programador, a criação da pasta e dos nomes fica a cargo do programa.

- Ao enviar os *logs* para o servidor:

  Antes: Depende de um *Resource String* que contém os nomes dos arquivos, tem de ser declarado pelo programador, ou seja, fica na aplicação estendida, fazendo com

[7] Mais informações sobre este recurso podem ser encontrados em: https://developer.android.com/guide/topics/resources/string-resource. Acesso em 15 de set. de 2018.

que o código do *framework* tenha que ser exclusivamente modificado para esta aplicação estendida.

Depois: Como os arquivos ficam armazenados em uma pasta específica (dentro da área restrita do *app*), todo arquivo que estiver dentro dessa pasta será enviado ao servidor. E o programador não tem que se preocupar com isso, pois está tudo encapsulado no *framework*.

- Criação da notificação de *status* do serviço:

  Antes: Ao criar a notificação é necessário que se conheça a classe principal da aplicação estendida (no caso uma *Activity*) para que quando a notificação for clicada a aplicação seja aberta, pois o "*.class*" é passado como parâmetro, fazendo novamente que o código do *framework* tenha que ser alterado para a aplicação estendida.

  Depois: Não é necessário saber o nome da classe principal, basta que se marque no arquivo de manifesto (*AndroidManifest*) a *Activity* com uma *action* adicional, denominada "LAUNCH_NOTIFICATION". Como funcionalidade adicional foi adicionado a possibilidade de o serviço poder ser encerrado diretamente da notificação.

- Alteração de valores para dados específicos da aplicação:

  Antes: Para que obtivesse um valor customizado, por exemplo para o IP do servidor de *logs*, deveria ser criado uma funcionalidade de gerenciamento de configurações, e teria que ser gravado (em um *SharedPreferences* preferencialmente), para que na próxima inicialização do aplicativo esses dados sejam carregados.

  Depois: Quando for necessário usar valores customizados basta que os valores de recurso do *framework* sejam sobrescritos (criar recursos com o mesmo nome) na área de recursos da aplicação estendida. Assim, só é necessário criar uma funcionalidade de gerenciamento de configurações se o desenvolvedor quiser ter a possibilidade de fazer as alterações em tempo de execução.

4.5 HoneypotLabsac: aplicabilidade

O trabalho anterior ainda adicionou a classificação de *Honeypots* monitores de portas de comunicação, que escutam essas portas e geram *logs* de toda comunicação feita.

Assim como o *framework* anterior o atual é de baixa interação, os serviços os quais o atacante irá interagir são simulados e os dados gerados dessa comunicação são coletados. E caso seja associado a um IDS pode alertar sobre tentativas de acesso não autorizado

Esses dados coletados geram arquivos de *log*, baseados na estrutura de Diebold (et al., 2005), cada linha contém a seguinte ordem de informações:

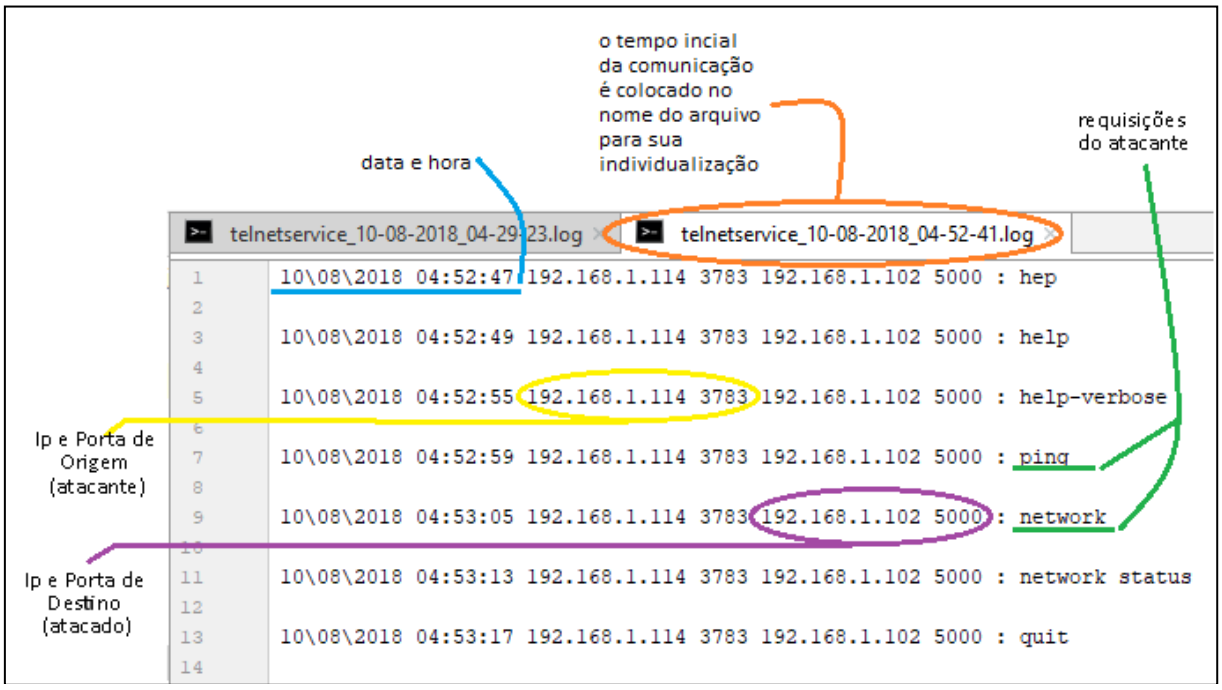
- Registro do tempo da comunicação;
- Endereço IP e porta de origem - atacante;
- Endereço IP e porta de destino – serviço;
- Requisições feitas pelo atacante, suas atividades durante a sessão.

Neste trabalho foi simulado o serviço de *Telnet*. Este serviço é uma adaptação baseada no funcionamento em emuladores *Android* e nos comandos ADB (*Android Debug Bridge*) (GOOGLE DEVELOPERS, 2018b), que é uma ferramenta de linha de comando para comunicação com emuladores e dispositivos reais.

Vale ressaltar que pra cada sessão com um atacante haverá um arquivo de *log* separado. Pode acontecer de existir várias sessões abertas ao mesmo tempo, portanto a distinção desses arquivos é dada com a criação de um ID para cada sessão na sua inicialização, e assim adicionando ao nome do arquivo.

Na Figura 8 pode-se ver a estrutura do *log* com um exemplo de interação:

FIGURA 8 - *Log* gerado pelo *Honeypot Labsac* 2.0

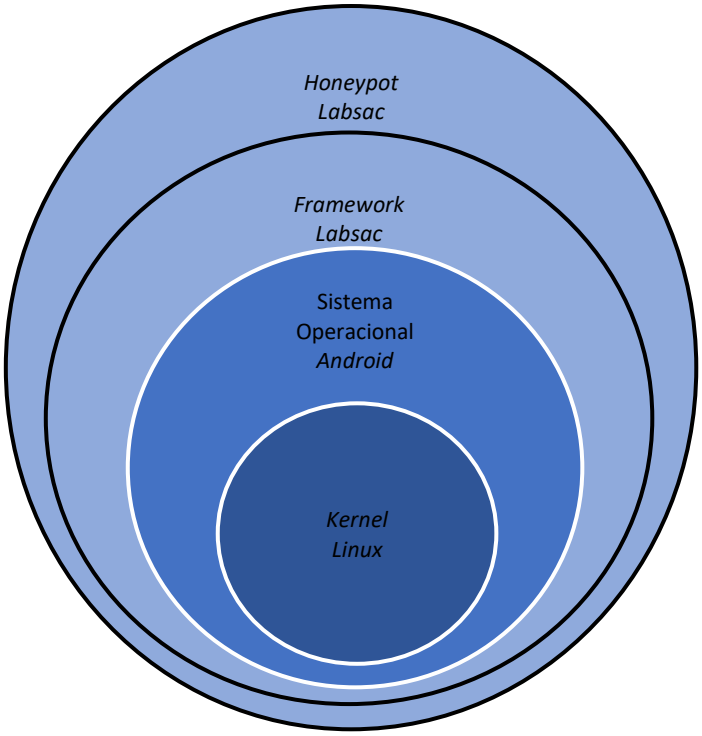


Fonte: O autor.

4.6 HoneypotLabsac: arquitetura

A arquitetura geral da nova versão permanece a mesma. A ela é acrescentada uma camada na arquitetura do *FrameworkLabsac*, apresentada na **seção 4.2**. A Figura 9 seguir apresenta esta arquitetura:

FIGURA 9 - Arquitetura do *HoneypotLabsac* 2.0



Fonte: O autor.

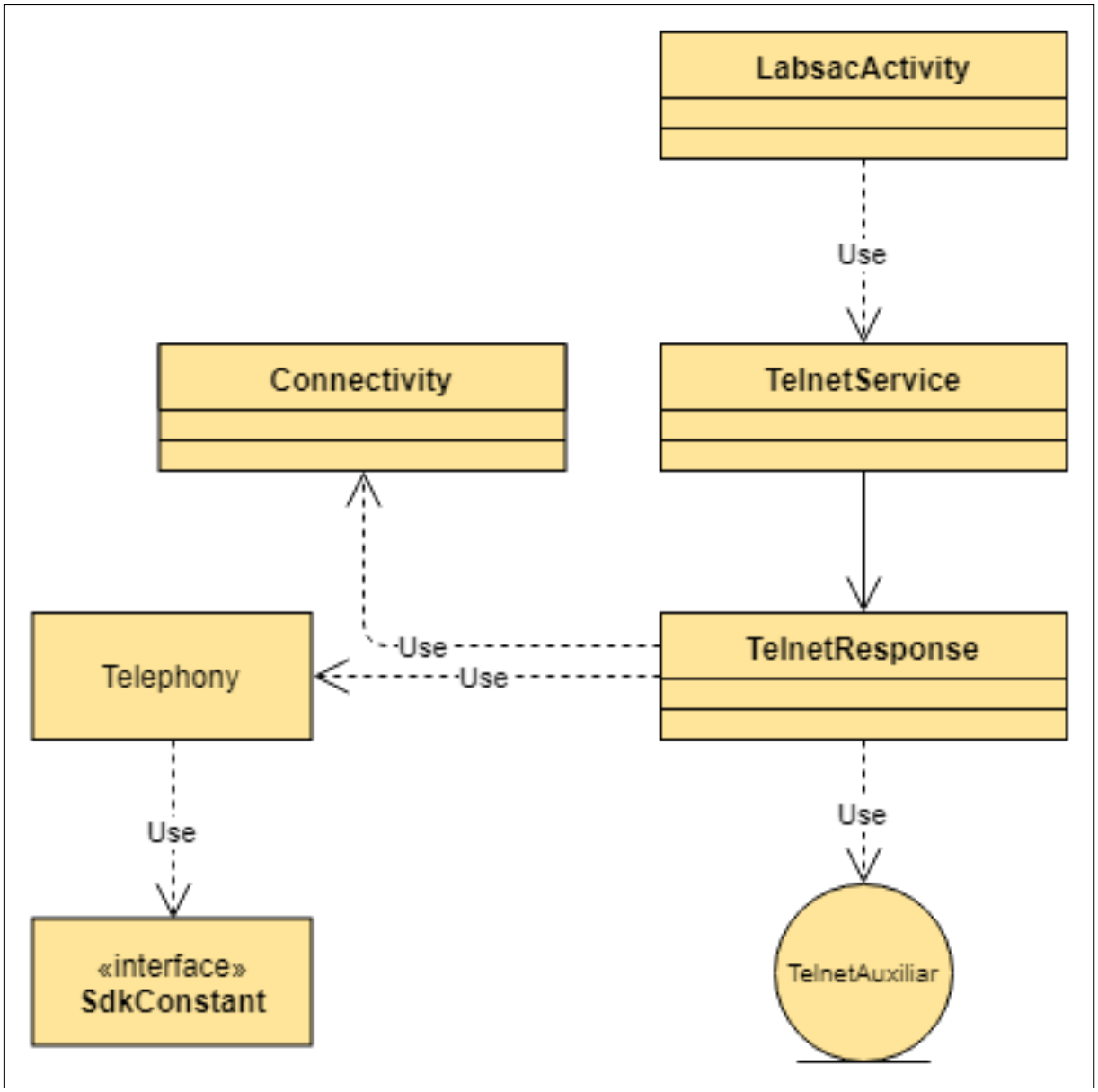
Detalhando a camada do *Honeypot Labsac* 2.0, pode-se destacar o componente *Telnet*, que tem o objetivo de simular o serviço de mesmo nome no sistema operacional *Android.*

4.7 HoneypotLabsac: implementação

Para desenvolver a camada de serviço também se utilizou a linguagem de programação *Kotlin*, com a IDE *Android Studio*. Deve ser destacado que é possível usar a linguagem *Java* para construir parte ou toda aplicação, pois as duas linguagens são interoperáveis entre si.

A seguir é mostrado a representação da estrutura de classes, seguido da explicação de suas funcionalidades:

FIGURA 10 - Diagrama de Classe da Aplicação



Fonte: O autor.

Tem-se o pacote *br.com.mario.honeypotLabsac.protocol.telnet*, onde ficam os componentes para criação do serviço *Telnet*:

- **SdkConstant**: é uma notação *Java* para criar anotações. Indica um valor constante para um campo que deve ser exportado para ser usado pelo SDK *tools*. É utilizada pela classe *Telephony*.
- **Telephony**: classe que manipula um *Provider* (Provedor), no caso o *Provider Telephony*, responsável por gerenciar operações do telefone, especialmente mensagens SMS e MMS. Usada para criar e ler mensagens *SMS* pelo serviço *Telnet*.
- **TelnetAuxiliar**: Em *Kotlin* é possível criar arquivos que não necessariamente precisam de classes encapsuladoras. Este arquivo contém classes, funções e

propriedades que servem para gerenciar o uso dos Sensores do dispositivo, que são utilizados na simulação do serviço *Telnet*.

- **TelnetResponse**: classe que herda de *ProtocolResponse*. Responsável por implementar o gerenciamento da comunicação com o atacante, respondendo de acordo com os comandos recebidos.
- **TelnetService**: classe que herda de *GeneralService*. Responsável por criar o serviço propriamente dito. Aqui são definidas algumas constantes para o bom funcionamento, como o nome para o serviço, um ícone para a notificação que será criada, e o mais importante é associado aqui o *Response* correspondente.

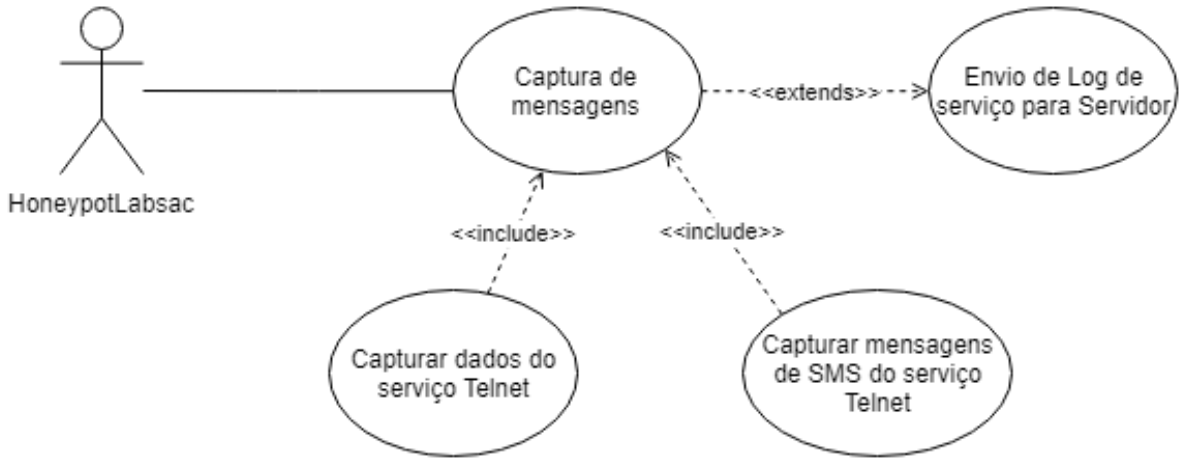
O pacote mais externo, *br.com.mario.honeypotLabsac*, possui as seguintes classes:

- **LabsacActivity**: classe que herda de *HoneypotActivity*. Responsável por criar o gerenciamento do *Honeypot* contando com uma interface gráfica. É a classe inicial e principal.
- **TranslucentActivity**: classe que herda de *HoneypotActivityTranslucent*. Responsável por criar o gerenciamento do *Honeypot*, mas sem uma interface. Uma classe para feita apenas para exemplificar essa funcionalidade.

4.8 HoneypotLabsac: caso de uso

Na Figura 11 podemos ver o diagrama de Caso de Uso para captura de dados. Toda interação com o serviço *Telnet* ou o recebimento de SMS através da conexão *Telnet* gera arquivos de *log*. Estes arquivos são primeiramente armazenados na seção privada da aplicação na memória interna do aparelho, e depois são enviadas para um servidor confiável.

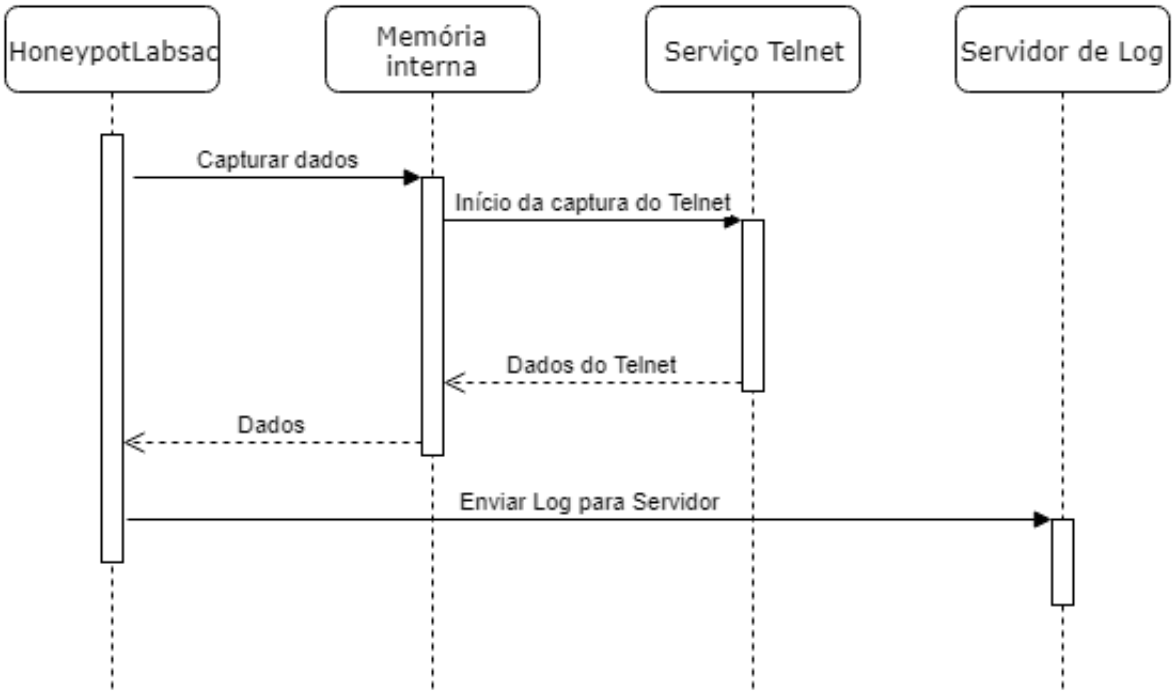
FIGURA 11 - Caso de Uso componente Captura



Fonte: O autor.

Na figura 12 é mostrado a sequência do processo de gerar *log* com a interação das classes ao passar do tempo:

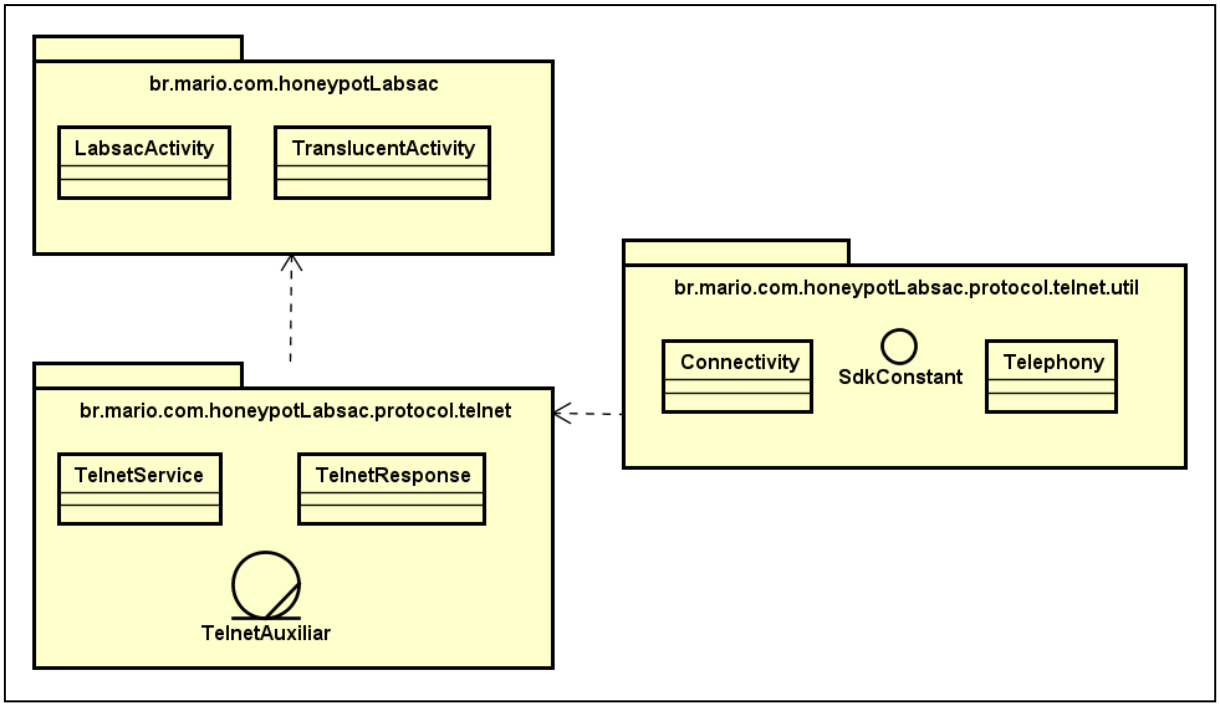
FIGURA 12 - Diagrama de Sequência da geração de *Log*



Fonte: O autor.

A seguir temos o detalhamento da estrutura de pacote da aplicação:

FIGURA 13 - Diagrama de Pacote do *HoneypotLabsac* 2.0

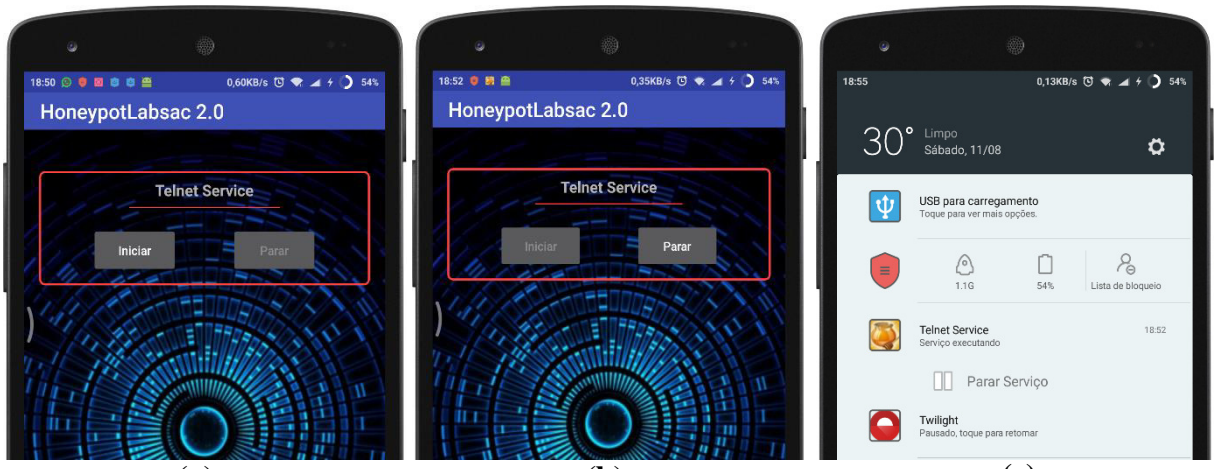


Fonte: O autor.

4.9 HoneypotLabsac: uso do *Honeypot Labsac* 2.0

Aqui é mostrado como está a interface para gerenciar o serviço *Telnet* na aplicação de exemplo criada:

FIGURA 14 - Tela inicial e controle do serviço Telnet



(a) **(b)** **(c)**

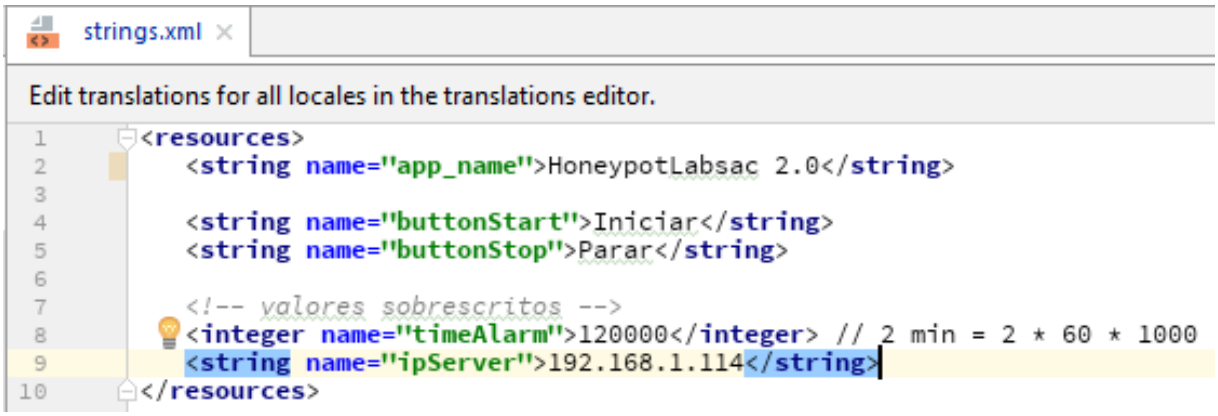
Fonte: O autor.

A figura 14 é composta por três imagens relacionadas à interface da aplicação de teste criada. São elas: a) a tela inicial antes do serviço ser inicializado, é uma tela simples, apenas com comandos de iniciar e parar o serviço; b) temos a tela após o serviço ser inicializado, nota-se que aparece a notificação do serviço na barra de notificações; c) o detalhe da notificação

do serviço *Telnet*, que além de indicar que o serviço está executando contém um botão de ação para interromper a execução do mesmo:

Para a configuração dos valores de IP e porta do servidor, foi usado a sobreposição no arquivo de recurso do *Android strings.xml*. Nele é sobrescrito apenas o IP, pois o valor para a porta é reutilizado do valor já definido no *Framework* 2.0. O valor para o tempo do intervalo entre disparos do alarme que chama o envio dos *logs* para o servidor também é sobrescrito neste arquivo. Como podemos ver na Figura 15:

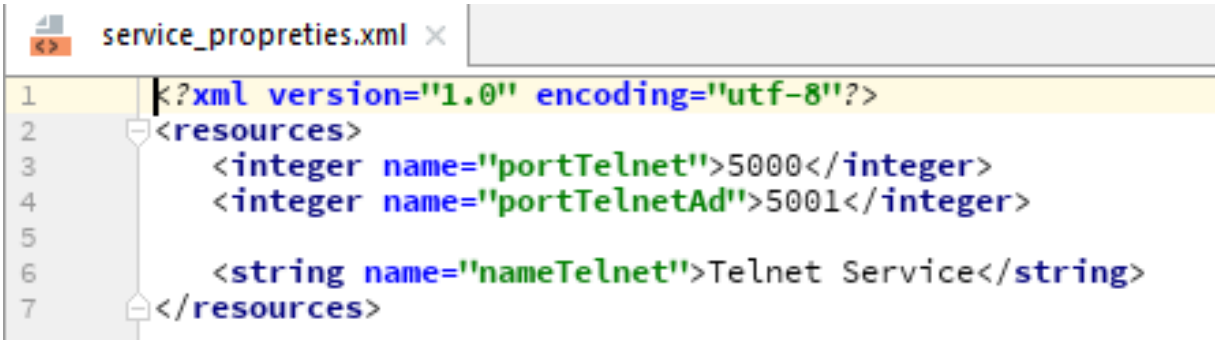
FIGURA 15 – Recursos sobrescritos para IP do servidor e tempo do alarme disparador do envio do logo para o servidor



Fonte: O Autor

E para os valores do serviço *Telnet* em específico criou-se um arquivo de recurso próprio chamado *service_properties.xml* (pode-se criar um arquivo para cada serviço, se preferir) e esses valores são resgatados em tempo de execução quando o aplicativo é aberto.

FIGURA 16 - Recursos para serviço *Telnet*



Fonte: O autor.

5 CONCLUSÃO

Com o refatoramento do "*HoneypotFramework*", obtivemos o "*HoneypotFramework 2.0*", um trabalho que pretende contribuir para a área de Segurança de Redes aplicada a dispositivos móveis baseados em *Android*. A refatoração expõe uma classificação do tipo virtual e escutador de portas, melhorando a estrutura e uso do *HoneypotFramework*, sendo agora possível classificá-lo como *framework* de caixa-branca e de Aplicação. Essa classificação permite criar e configurar um *Honeypot* para obter seus resultados mais facilmente, além de torná-lo independente. Tais características podem servir de exemplo para novas melhorias em serviços dessa área de estudo, seja em recursos já existentes ou na criação de novos, como Servidor HTTP, gerenciamento do SMS, Servidor FTP, etc.

A validação do *Framework* 2.0 foi feita utilizando uma simulação do serviço do *Telnet*, o qual deu o suporte para a construção de um *Honeypot*, com geração de *logs*, fornecendo ferramentas que possibilitam as configurações necessárias.

Conclui-se que os principais aspectos melhorados foram: estrutura, uso, obtenção de resultados e independência do código do *framework*, estes elementos são cruciais na conquista de maior eficiência na elaboração de um serviço de *Honeypot*, portanto, são chaves do processo de otimização desse tipo de serviço. Conhecendo esses pontos, exibe-se a possibilidade de ampliação deste estudo, via novos estudos e necessidades do campo de segurança de dispositivos móveis *Android*.

REFERÊNCIAS

ANDROID DEVELOPERS. **Kotlin and Android**. [2016?] .Disponível em: <https://developer.android.com/kotlin/>. Acesso em: 13 jan. 2018.

ANDROID OPEN SOURCE PROJECT. **The Android Source Code**. 2017. Disponível em: <http://source.android.com/source/index.html>. Acesso em: 3 jul. 2018.

ANDROID OPEN SOURCE PROJECT. **Security**. 2018a. Disponível em: <https://source.android.com/security/>. Acesso em: 20 nov. 2017.

ANDROID OPEN SOURCE PROJECT. **Application Sandbox**. 2018b. Disponível em: <https://source.android.com/security/app-sandbox>. Acesso em: 23 ago. 2018.

APACHE FOUNDATION. **Apache Is Open**. Disponível em: <https://blogs.apache.org/foundation/entry/apache-is-open>. Acesso em: 29 jun. 2018.

BALMAS, Y. Mobilie Network Securuty: Availability Risks on Mobile Networks. 2013. Disponível em: <https://security.radware.com/uploadedFiles/Resources_and_Content/Attack_Tools/Mobile_Networks_Security_Research_Paper.pdf>. Acesso em: 26 out. 2018.

BRAGA, A. M.; NASCIMENTO, E. N. Do; PALMA, L. R. Da; ROSA, R. P. Introdução à Segurança de Dispositivos Móveis Modernos – Um Estudo de Caso em Android. **Minicursos do XII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais**, 2012. Disponível em: <https://www.researchgate.net/profile/Alexandre_Braga2/publication/273458482_Introducao_a_Seguranca_de_Dispositivos_Moveis_Modernos-Um_Estudo_de_Caso_em_Android/links/5655d51808aeafc2aabe2c6b.pdf>. Acesso em: 29 jun. 2018.

CHESWICK, B. **An Evening with Berferd In Which a Cracker is Lured, Endured, and Studied**. Disponível em: <http://www.cheswick.com/ches/papers/berferd.pdf>. Acesso em: 29 jun. 2018.

COHEN, F. **The Use of Deception Techniques: Honeypots and Decoys**. Disponível em: <http://adams.all.net/journal/deception/Deception_Techniques_.pdf>. Acesso em: 28 jun. 2018.

Definition of Framework. [20-?]. Disponível em: <https://www.merriam-webster.com/dictionary/framework>. Acesso em: 1 jul. 2018.

DIEBOLD, P.; HESS, A.; SCHÄFER, G. A Honeypot Architecture for Detecting and Analyzing Unknown Network Attacks. In: **Kommunikation in Verteilten Systemen (KiVS)**. Berlin/Heidelberg: Springer-Verlag, 2005. p. 245–255. Berlin/Heidelberg: Springer-Verlag. Disponível em: <https://www.researchgate.net/profile/Guenter_Schaefer2/publication/221500107_A_Honeypot_Architecture_for_Detecting_and_Analyzing_Unknown_Network_Attacks/links/54140db40cf2fa878ad3e3bb/A-Honeypot-Architecture-for-Detecting-and-Analyzing-Unknown-

Network-Att>. Acesso em: 5 jul. 2018.

ECLIPSE FOUNDATION. **About the Eclipse Foundation**. 2018. Disponível em: <http://www.eclipse.org/org/>. Acesso em: 5 de set 2018.

EHRINGER, D. **The Dalvik Virtual Machine Architecture**. Disponível em: <http://www.davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf>. Acesso em: 5 jul. 2018.

FAYAD, M.; SCHMIDT, D. C. Object-oriented application frameworks. **Communications of the ACM**, v. 40, n. 10, p. 32–38, 1997. Disponível em: <https://pdfs.semanticscholar.org/629d/faa3353f1e830d6d377ec11cf0c0550f608a.pdf>. Acesso em: 11 jul. 2018.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns: Elements of Reusable Software**. Disponível em: <https://sophia.javeriana.edu.co/~cbustaca/docencia/DSBP-2018-01/recursos/Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides-Design Patterns_ Elements of Reusable Object-Oriented Software -Addison-Wesley Professional (1994).pdf>. Acesso em: 10 jul. 2018.

GERO, J. Design Prototypes: A Knowledge Representation Schema for Design. **AI Magazine**, v. 11, n. 4, p. 26, 1990. Disponível em: <http://www.aaai.org/ojs/index.php/aimagazine/article/view/854>. Acesso em 10 de jul. de 2018.

GOOGLE DEVELOPERS. **Platform Architecture**. Disponível em: <https://developer.android.com/guide/platform/index.html>. Acesso em: 3 jan. 2018.

GOOGLE DEVELOPERS. **Conheça o Andrid Studio**. Disponível em: <https://developer.android.com/studio/intro/?hl=pt-br>. Acesso em: 12 jul. 2018a.

GOOGLE DEVELOPERS. **Android Debug Bridge (adb)**. Disponível em: <https://developer.android.com/studio/command-line/adb>. Acesso em: 8 ago. 2018b.

HEISS, J. J. **The Advent of Kotlin: A Conversation with JetBrains' Andrey Breslav**. Disponível em: <http://www.oracle.com/technetwork/articles/java/breslav-1932170.html>. Acesso em: 29 jun. 2018.

HOEPERS, C.; STENDING-JENSEN, K.; CHAVES, M. H. P. C. **Honeypot e Honeynets: Definições e Aplicações**. Disponível em: <https://www.cert.br/docs/whitepapers/honeypots-honeynets/>. Acesso em: 21 jan. 2018.

IDC: ANALYZE THE FUTURE. **Smartphone OS**. Disponível em: <https://www.idc.com/promo/smartphone-market-share/os>. Acesso em: 14 jan. 2017.
JETBRAINS. **About**. 2018. Disponível em: <https://www.jetbrains.com/company/>. Acesso em: 12 jul. 2018.

JETBRAINS. **Kotlin Language Documentation**. [201-?]. Disponível em: <https://kotlinlang.org/docs/kotlin-docs.pdf>. Acesso em: 08 set. 2018.

JOHNSON, R. E. Frameworks = (components + patterns). **Communications of the ACM**, v. 40, n. 10, p. 39–42, 1 out. 1997. Disponível em: <http://doi.acm.org/10.1145/262793.262799>. Acesso em: 30 jun. 2018.

JONES, B. **Here Are the Most Common Android Viruses**. Disponível em: <https://www.psafe.com/en/blog/common-android-viruses/>. Acesso em: 21 jan. 2018.

LANDIN, N.; NIKLASSON, A.; BOSSON, G.; REGNELL, B. Development of Object-Oriented Frameworks. **Department of Communication System. Lund Institute of Technology, Lund University. Lund, Sweden**, 1995. Disponível em: <http://www.dsc.ufcg.edu.br/~jacques/cursos/map/recursos/developing-frame.pdf>. Acesso em: 11 jul. 2018.

LI, H.; VAN KATWIJK, J.; LEVY, A. M. **Reuse of software design and software architecture**. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=5F9E269A2D22841D74118C4CCACC9B65?doi=10.1.1.133.6414&rep=rep1&type=pdf>. Acesso em: 11 jul. 2018.

MEYER, M. **A história do Android [Atualizado Android Oreo 8.1]**. Disponível em: <https://www.oficinadanet.com.br/post/13939-a-historia-do-android>. Acesso em: 3 jul. 2018.

NOVICE'S BLOG. **Android Architecture**. Disponível em: <https://novicesuresh.com/tag/android-architecture-2/>. Acesso em: 2 jul. 2018.

NOVICE'S BLOG. Android Architecture. Disponível em: <https://novicesuresh.com/tag/android-architecture-2/>. Acesso em: 2/7/2018.

OLIVEIRA, V. B. De. **HoneypotLabsac : um framework de honeypot virtual para o android**. 2012. Universidade Federal do Maranhão, 2012. Disponível em: <http://www.tedebc.ufma.br/tde_arquivos/10/TDE-2012-11-28T093437Z-692/Publico/dissertacao Vladimir Bezerra.pdf>. Acesso em 13 de jan. de 2018.

OPEN HANDSET ALLIANCE. **OHA Overview**. 2017. Disponível em: <http://www.openhandsetalliance.com/oha_overview.html>. Acesso em: 3 jan. 2017.

OPEN HANDSET ALLIANCE. **Android Overview**. 2017. Disponível em: <https://www.openhandsetalliance.com/android_overview.html>. Acesso em: 3 jul. 2018.

ORACLE. **Obtenha Informações sobre a Tecnologia Java**. [20-]. Disponível em: <https://www.java.com/pt_BR/about/>. Acesso em: 5 de set. 2018.

SCOTA, D. F.; ANDRADE, G. E. de; XAVIER, R. da C. **Configuração de Rede Sem Fio e Segurança no Sistema Operacional Android**. Disponível em: <https://www.ppgia.pucpr.br/./Daniel Fernando Scota - Artigo.pdf>. Acesso em: 21 jan. 2018.

SHAFIROV, M. **Kotlin On Android. Now Official**. Disponível em: <https://blog.jetbrains.com/kotlin/2017/05/kotlin-on-android-now-official>. Acesso em: 29 jun. 2018.

SIGNIFICADOS. **Significado de Ataque Cibernético**. Disponível em: <https://www.significados.com.br/ataque-cibernetico>. Acesso em: 21 jan. 2018.

SILVA, A. L. da. **Modelo de IDS para Usuários de Dispositivos Móveis**. Disponível em: <https://tedebc.ufma.br/jspui/bitstream/tede/335/1/Aline lopes.pdf>. Acesso em: 4 jul. 2018.

SINGH, M.; VANEET. Linux Kernel Memory Protection (ARM). **International Journal of Computer Science and Information Technologies (IJCSIT)**, v. 5, n. 4, p. 5869–5871, 2014. Disponível em < http://ijcsit.com/docs/Volume%205/vol5issue04/ijcsit20140504225.pdf>. Acesso em: 13 de set. de 2018.

SPITZNER, L. **Honeypots: tracking hackers**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., v 1ª, 2002.480 p. ISBN 0-321-10895-7. Disponível em: <http://www.it-docs.net/ddata/792.pdf>. Acesso em: 05 de jul. de 2018.

STACK OVERFLOW. **Most Loved, Dreaded and Wanted Languages**. Disponível em: <https://insights.stackoverflow.com/survey/2018#most-loved-dreaded-and-wanted>. Acesso em: 29 jun. 2018.

STOLL, C. **Stalking The Wily Hacker**. Disponível em: <http://pdf.textfiles.com/academics/wilyhacker.pdf>. Acesso em: 1 jul. 2018.

T-CONSULTING SPREITZENBARTH. **Current Android Malware | forensic blog**. Disponível em: <http://forensics.spreitzenbarth.de/android-malware/>. Acesso em: 22 nov. 2017.

THE HONEYPOT PROJECT. 2018. Disponível em: <https://www.honeynet.org/about>. Acesso em: 5 jul. 2018.

APÊNDICE

APÊNDICE A – PASSO A PASSO DE COMO CONSTRUIR UM SERVIÇO UTILIZANDO O FRAMEWORK PROPOSTO

I. Primeiramente adiciona-se o .jar ou o módulo do *FrameworkLabsac* no seu projeto;

II. Deve-se criar um protocolo para o serviço, portanto vamos é criado um pacote '*protocol*';

III. Dentro do pacote é criada uma classe com um nome que lembre o serviço, um nome completo como exemplo: *protocol*. Telnet;

IV. Esta classe irá estender de *ProtocolResponse* e deve implementar a função abstrata *respond()*, que é de onde as respostas ao atacante serão enviadas;

V. Para efetuar o envio de fato da resposta, ao final de cada resposta construída deve-se chamar a função *flush()*, para ser enviada pelo streaming;

VI. Com a classe de protocolo criada, parte-se para criar o serviço em si, que será uma classe herdeira de *GeneralService*, e deve-se implementar a função *onStartCommand()*;

VII. Dentro de *onStartCommand()* serão inicializadas algumas informações para o serviço e sua notificação executarem. Para a notificação deve ser passado nome, um id, opcionalmente um ícone customizado para o serviço;

VIII. Agora, para criar o ponto de entrada, que no caso é uma tela inicial que conterá as configurações e controle da aplicação a classe *HoneypotActivity* deve ser estendida. Na classe filha será necessário seguir os seguintes passos para inicialização e execução correta dos componentes: a) sobrescrever a função *configureVariables()*, que serve para inicializar itens pré-criação de *Context*, e inicializará *timeInMilliSeconds* (valor padrão 4500) que que é o tempo para repetição do envio dos logs para servidor, inicializar a variável *layout_main* que conterá o valor de recurso de layout criado

IX. sobrescrever a função *startComponents()*, que serve para inicializar variáveis pós-criação do *Context*, e dentro dela inicializar variáveis booleanas para controle de execução, variáveis inteiras para guardar as portas para o socket de cada serviço existente. sobrescrever *startServiceImages()*, que inicializará cada variável de imagem de serviço (instâncias de *ServiceImage*).

X. sobrescrever *updateLayout()*, será chamado a cada inicialização do aplicativo. E deve ser chamado quando houver mudança de execução dos serviços que reflita na interface. Se utiliza das variáveis booleanas que guardam o estado de execução dos serviços. E

deve sempre chamar primeiramente a sua função correspondente na classe pai, cuja finalidade é atualizar os valores booleanos.

XI. Implementar *updateServices()*, sempre chamada pela função *updateLayout()* da classe pai, atualiza as variáveis booleanas que monitoram a execução dos serviços. Faz isso se utilizando da função estática *checkPort()* da classe *HoneypotStarter*.

XII. para controlar a inicialização e finalização dos serviços são utilizados os métodos *start()* e *stop()*. São do tipo ‘final’ e estão implementadas na classe pai, *HoneypotActivity*.