Android OS: Um estudo sobre engenharia reversa em aplicações do sistema

Leonardo de Almeida Silva de Andrade[1]

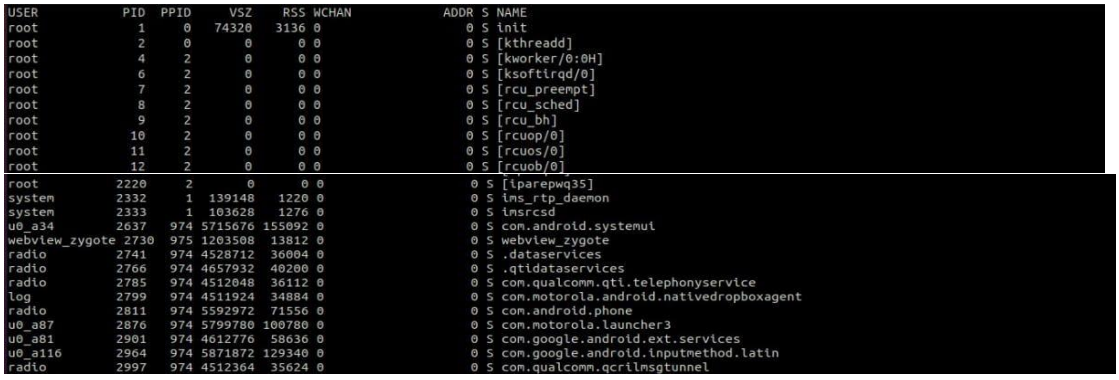
[1]Grupo de Resposta a Incidentes de Segurança - UFRJ, Rio de Janeiro, RJ, Brasil

A partir do crescente desenvolvimento e utilização de aplicações mobile, vê-se que cada vez mais as informações estão sendo alocadas em aparelhos celulares. Documentos de identificação, contas em redes sociais, senhas, números de cartão de crédito, conversas privadas, entre outros, já estão concentrados na palma da mão do usuário. Nesse sentido, entende-se que a Segurança da Informação é um tópico inerente. Alguns casos de ataques vão desde o vazamento das conversas do *Telegram* de políticos brasileiros até o episódio onde usuários de *Pokemón GO* estudaram o funcionamento da aplicação para controlar a geolocalização no jogo. Com isso, vê-se a necessidade do estudo e compreensão do funcionamento das aplicações Android, assim como suas vulnerabilidades e mecanismos de proteção.

Aplicações Android | Segurança da Informação |

Correspondence: *leonardo_andrade@poli.ufrj.br*

Introdução. O sistema operacional Android pode ser entendido como tendo dois lados distintos: o kernel do Linux reduzido e modificado e uma máquina virtual de aplicações que executa aplicativos estilo Java. Há algumas diferenças entre a linha principal do kernel Linux e do kernel Android. Dentre as fundamentais, podemos destacar que no Linux convencional as aplicações iniciadas por um usuário são executadas dentro do contexto deste usuário. Este modelo depende que um software malicioso não seja instalado porque não há proteção contra acesso à arquivos pertencentes ao mesmo usuário que está executando. Por outro lado, cada aplicativo instalado no dispositivo Android é assinado pelo seu próprio identificador único de usuário (*unique user identifier* – UID) e identificador de grupo (*group identifier* - GID).



```
USER           PID  PPID     VSZ    RSS WCHAN              ADDR S NAME
root             1     0   74320   3136 0                    0 S init
root             2     0       0      0 0                    0 S [kthreadd]
root             4     2       0      0 0                    0 S [kworker/0:0H]
root             6     2       0      0 0                    0 S [ksoftirqd/0]
root             7     2       0      0 0                    0 S [rcu_preempt]
root             8     2       0      0 0                    0 S [rcu_sched]
root             9     2       0      0 0                    0 S [rcu_bh]
root            10     2       0      0 0                    0 S [rcuop/0]
root            11     2       0      0 0                    0 S [rcuos/0]
root            12     2       0      0 0                    0 S [rcuob/0]
root          2220     2       0      0 0                    0 S [lparepwq35]
system        2332     1  139148   1220 0                    0 S ims_rtp_daemon
system        2333     1  103628   1276 0                    0 S imsrcsd
u0_a34        2637   974 5715676 155092 0                    0 S com.android.systemui
webview_zygote 2730  975 1203508  13812 0                    0 S webview_zygote
radio         2741   974 4528712  36004 0                    0 S .dataservices
radio         2766   974 4657932  40200 0                    0 S .qtidataservices
radio         2785   974 4512048  36112 0                    0 S com.qualcomm.qti.telephonyservice
log           2799   974 4511924  34884 0                    0 S com.motorola.android.nativedropboxagent
radio         2811   974 5592972  71556 0                    0 S com.android.phone
u0_a87        2876   974 5799780 100780 0                    0 S com.motorola.launcher3
u0_a81        2901   974 4612776  58636 0                    0 S com.google.android.ext.services
u0_a116       2964   974 5871872 129340 0                    0 S com.google.android.inputmethod.latin
radio         2997   974 4512364  35624 0                    0 S com.qualcomm.qcrilmsgtunnel
```

Fig. 1-2. Uma saída *snippet* do comando "*adb shell ps*", note que as aplicações rodam como usuários diferentes.

De suma importância, temos a Dalvik Virtual Machine (DVM), a máquina virtual que foi desenhada exclusivamente para a plataforma Android. Mais leve que uma VM Java, a VM Dalvik é baseada em registradores, o que a torna mais rápida e além disso economiza a bateria do aparelho. O sucessor da DVM é o ART (Android Runtime), que traz possui melhor performance em relação à anterior. Por exemplo, enquanto na DVM o código fonte é compilado durante a sua a execução, a ART usa a abordagem Ahead-of-Time (AOT), o que significa que ela compila o código fonte antes da sua execução, isto implica num ganho de desempenho e de bateria nos aparelhos Android.

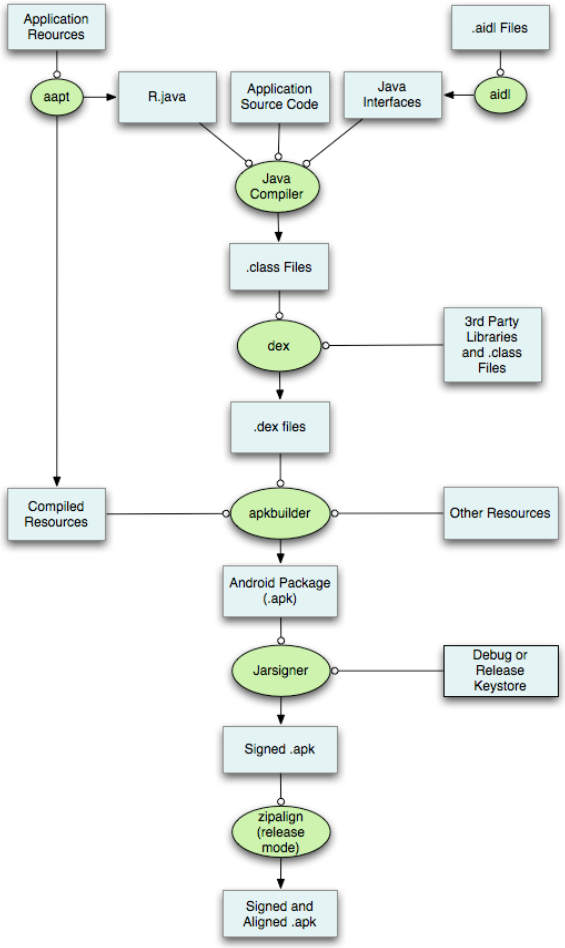


Fig. 3. *Processo de Construção do Android*

É interessante entender como funciona aquilo em que se deseja realizar a engenharia reversa, por isso vamos entender o processo de construção e compilação de um APK (aplicação) Android.

Primeiro os compiladores reúnem o código fonte da aplicação, as classes Java, os recursos e a interface e converte-os em arquivos DEX que contêm o bytecode compatível com o Dalvik e em recursos compilados.

A seguir o APK Manager reúne os arquivos DEX e os recursos compilados em um único arquivo *.apk* (APK = *Android Application Pack*). Porém esse APK ainda não pode ser executado porque não possui um certificado válido, é necessário assiná-lo. Será utilizado o *jarsigner* para esse processo, havendo a opção de assinar com uma keystore de *debug* (usado quando a aplicação é apenas um teste ou depuração, e não um produto final) ou uma keystore de lançamento (usado para assinar produtos finais).

Por fim, é usado o *zipalign* para otimizar o espaço de memória ocupado pela aplicação.

Pacotes Android. Neste tópico faremos uma análise dos pacotes que estão presentes num arquivo APK.

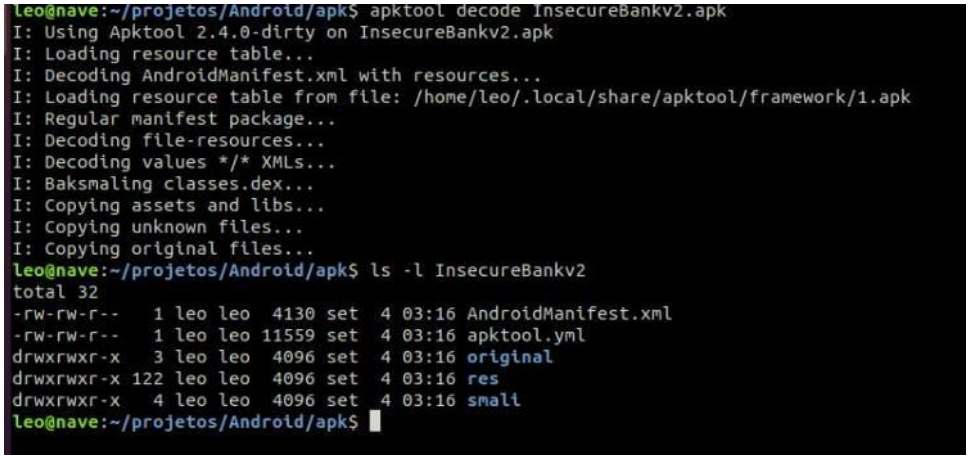


Fig. 4. Decodando a aplicação e listando os arquivos do InsecureBankv2, uma aplicação vulnerável desenvolvida exclusivamente para *pentest* e aprendizado.

Note que utiizamos o *apktool* para decodificar a aplicação. O *apktool* é uma ferramenta para fazer engenharia reversa em arquivos APK, será de grande utilidade nesse procedimento.

O *apktool* criou um diretório contendo alguns arquivos reunidos a partir da decodificação do APK. Vamos entender o que cada um significa.

res/: Esta pasta é usada para guardar os valores para os recursos que serão usados pela aplicação, incluindo características de cores, estilos, dimensões, entre outros.

original/: Pasta que contém o META-INF e o AndroidManifest.xml original.

original/META-INF/: Esta pasta contém o certificado do aplicativo e arquivos que contêm uma lista de inventário de todos arquivos incluídos no arquivo zip e seus hashes.

AndroidManifest.xml: O Android Manifest é um arquivo que contém as informações fundamentais para compiladores, para o sistema Android e para o Google PlayStore. No manifesto estão listadas informações como o nome do pacote, as permissões requisitadas pela

aplicação (como acesso à câmera, internet, enviar SMS), as *activities*, filtro de intents, entre outros. Teremos uma seção exclusiva para falar sobre o arquivo manifesto.

smali/: Esta é a pasta importante, o apktool decodifica o código Java compilado na forma de arquivos *.smali*. Podemos comparar a linguagem Smali com o Assembly. Ao fazer o disassembler de um programa escrito em C por exemplo, o que obtemos é a linguagem Assembly de baixo nível. De modo semelhante, o apktool decompila um arquivo DEX para o Smali. Esses arquivos em Smali são responsáveis pela funcionalidade da aplicação e manuseá-los cria impactos diretos no app. Podemos visualizar e editar o código Smali com um simples editor como o Vim por exemplo.

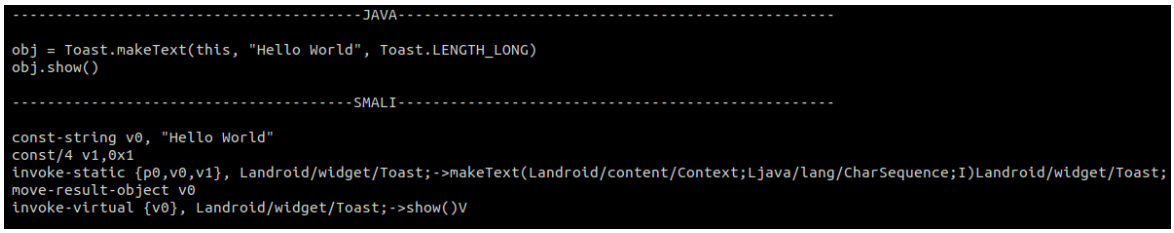


Fig.5. Mensagem Toast *Hello World* em Java e Smali

Entendendo o AndroidManifest.xml.

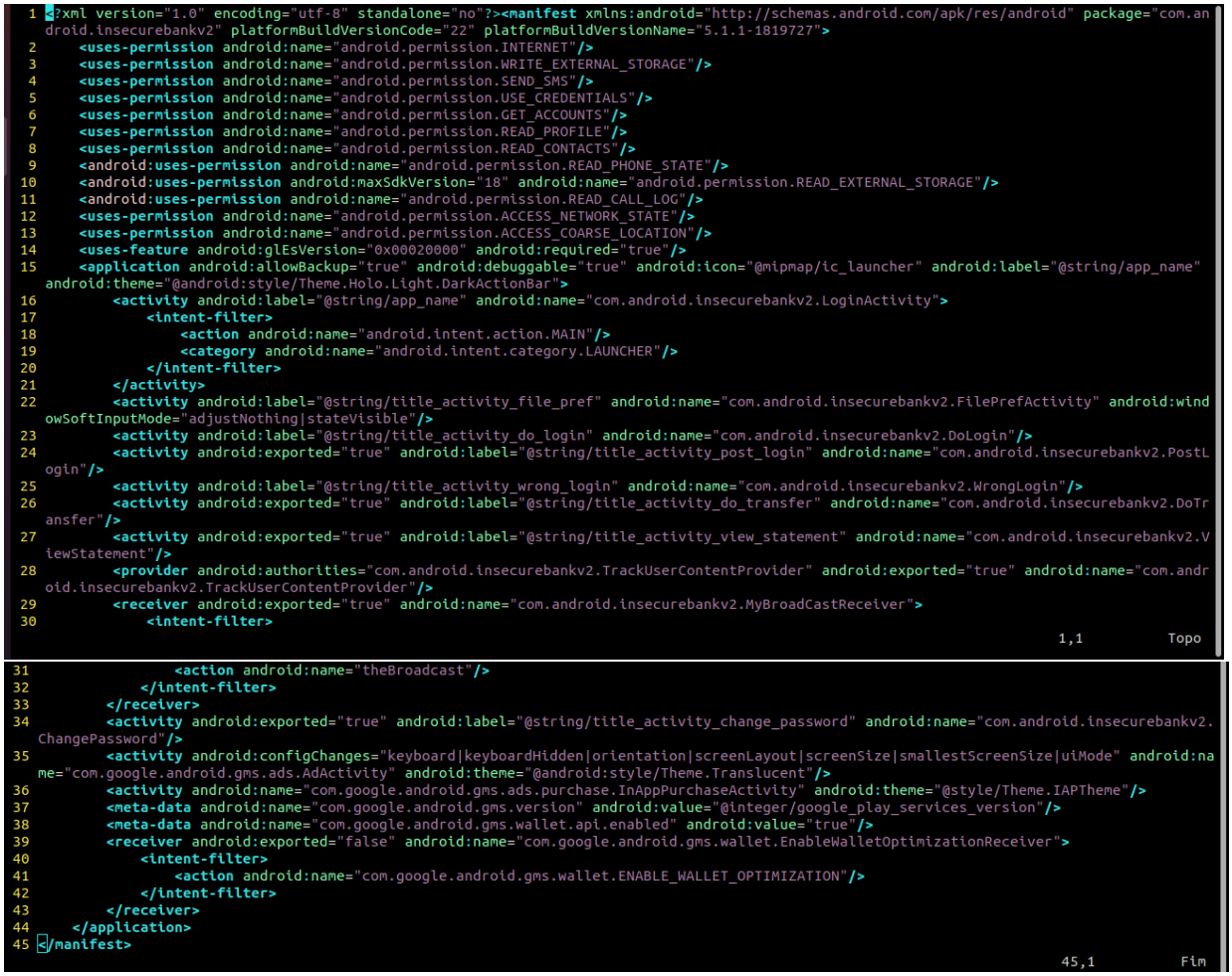


Fig. 6-7 AndroidManifest.xml do InsecureBankv2

Um dos fundamentos da segurança do Android é que cada aplicação roda em sua própria *sandbox*. Todavia, pode-se pontuar alguns casos onde as aplicações podem compartilhar os mesmos processos, e isto ocorre quando é declarado o atributo "*sharedUserId*" no AndroidManifest.xml. Isso caracteriza uma vulnerabilidade em potencial, por exemplo, se duas aplicações X e Y possuem aquele atributo definidos com o mesmo valor, eles compartilharão mesmo ID, desde que os conjuntos de certificados deles sejam idênticos, ou seja, foram assinados pela mesma chave privada do desenvolvedor. Apps com o mesmo ID podem acessar recursos e dados um do outro e, como foi dito anteriormente, serem executados no mesmo processo.

Sobre os pacotes Android, temos que a convenção de nomenclatura para eles deve ser totalmente em letras minúsculas e o nome de domínio reverso da Internet da organização que o desenvolveu. Por exemplo, a empresa "Jogos Android" desenvolveu a aplicação "corrida de moto", o nome do pacote deverá ser *com.jogosandroid.corridademoto*. Isso garante a organização e separação do grupo e que haja apenas um pacote com esse nome, o qual será

declarado no atributo *package* do manifesto e usado para resolver todos os nomes de classes relativas declaradas no arquivo de manifesto.

Para cada componente de aplicativo que for criado deverá haver um elemento XML declarado no manifesto. Com isso, é essencial entender o que é cada um desses componentes e que informação trazem no arquivo XML.

A. Activities.

De forma sucinta, é um ponto de interação com o usuário, como por exemplo uma caixa de texto. Todas as activities devem ser declaradas na tag *<activity>*, qualquer uma que não estiver neste arquivo não será reconhecida nem executada.

```
<activity android:label="@string/app_name" android:name="com.android.insecurebankv2.LoginActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
```

Fig. 8. Exemplo de Activity do InsecureBankv2.

Acima temos um exemplo que será muito útil na hora de depurar e fazer a engenharia reversa da aplicação. Note que na tag *<action>* ela é tida como MAIN e na tag *<category>* como LAUNCHER, isso significa que assim que o usuário iniciar o aplicativo esta será a primeira activity executada antes de qualquer outra. No atributo *android:name* percebe-se o nome dessa atividade.

B. Services.

São serviços, semelhantes às activities porém rodam em background e não possuem interação com o usuário. Como são executados em segundo plano, são utilizados para implementar operações de longa duração ou uma API de comunicação avançada que pode ser chamada por outros aplicativos.

C. Content Providers.

São "*storages*" que podem ser criados e acessados a partir de recursos internos da linguagem. Podem ser entendidos como armazenamento ou base de dados.

D. Broadcast Receivers.

Componentes desenvolvidos na aplicação com o intuito de interagir com o sistema.Ex.: quando se quer que o sistema exiba uma notificação quando um SMS chegar.

Além desses componentes, temos outros itens no arquivo manifesto:

E. Application Information.

Nome, versão, API Level, Sdk version.

F. User Permissions (<*uses-permission*>).

Permissões de usuário. O Android Manifest precisa especificar tudo que a aplicação quer ter acesso, como internet, câmera, contatos entre outros. Podemos diferenciar as permissões em dois tipos, as ***perigosas***, ou seja, que possam afetar a privacidade do usuário ou a operação do dispositivo, nesse caso o usuário deve concordar explicitamente em concedê-la. Por outro lado, as permissões ***normais*** são as que não apresentam muito risco à privacidade do usuário nem à operação do dispositivo, o sistema concederá automaticamente essas permissões.

Ex.: *android.permission.CAMERA, android.permission.READ_CONTACTS*.

G. Components Resources.

Como a aplicação armazena ícones, arquivos xml. Ex.: uma aplicação internacionalizada, ou seja, tem uma interface em inglês, português, espanhol, entre outros, é possível especificar a tradução do app em arquivos xml internos ou tabelas de string que ficam armazenados nos *components resources.*

H. <manifest...>

Principal tag do Android Manifest, não é "filha" de nenhuma outra tag e seu principal item é o *package*.

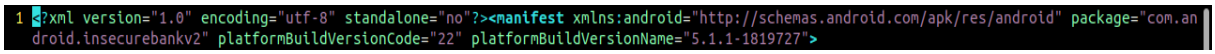


Fig. 9. Tag <manifest> do InsecureBankv2.

I. <application...>

Presente na tag <manifest>, deve-se procurar pelos atributos *android.allowBackup=[true|false]* (permite ou não que o aplicativo participe da infraestrutura de backup e restauração, se for *false*, nenhum backup ou restauração do aplicativo será realizado, mesmo por um backup completo do sistema que faria todos os dados do aplicativo serem salvos via *adb*. O valor padrão desse atributo é verdadeiro) e *android.debuggable=[true|false]* (define se aplicação é depurável, *true* para sim e *false* para não, o valor padrão é *false*).

J. Intent e Intent-Filters

Os *Intents* são recursos da aplicação e arquitetura Android que fornecem um meio de comunicação entre os componentes. Pode-se usar os *Intents* para chamar uma atividade, serviço, broadcast receivers, entre outros.

Ex.: Iniciando uma **Activity**: *startActivity()* | *startActivityForResult()*

Iniciando um **Service**: *startService()* | *bindService()*

Iniciando um **Broadcast**: *sendBroadcast()*

As propriedades de um Intent são:

- **Nome do componente** (apenas para Intents Explícitos). Ex.: *Activity.class.;*
- **Ação**. Ação genérica. Ex.: *ACTION_VIEW*, *ACTION_IMAGE_CAPTURE;*
- **Categoria**. Informação adicional. Ex.: *CATEGORY_LAUNCHER*;
- **Dados**. A URI que faz referência ao dado. Ex.: *www.google.com*
- **Extras**. Pares valores-chave com informação extra Ex.: EXTRA_EMAIL;
- **Sinalizadores**. Funcionam como metadados para a intent. Ex.: *FLAG_FROM BACKGROUND.*

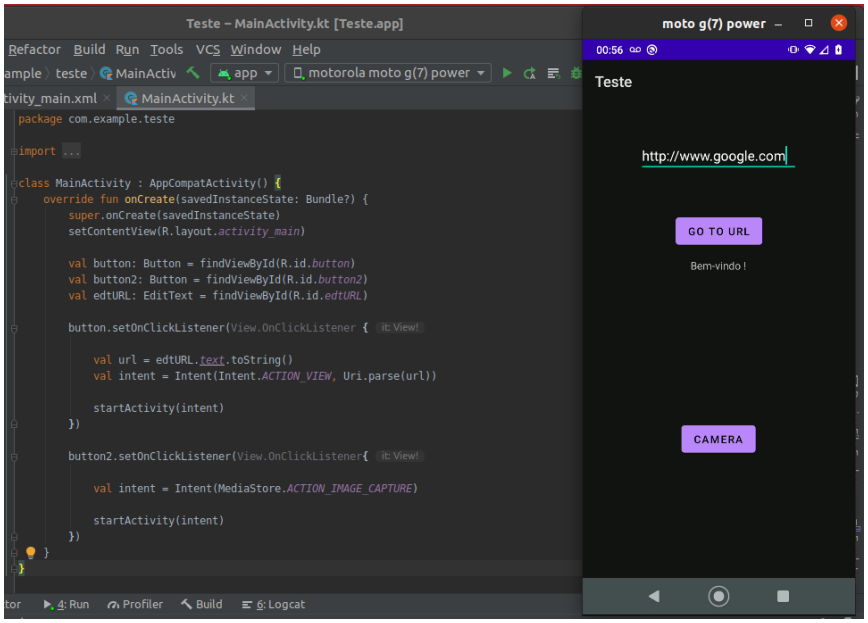


Fig.10. Exemplo da criação de dois Intents implícitos.

Na figura acima podemos destacar, o primeiro Intent indica ao sistema Android para exibir uma página web a partir do dado passado (no caso o endereço *www.google.com*) quando o *button* "Go to URL" for pressionado.

O segundo Intent é usado para chamar a câmera do aparelho quando o *button* "Camera" for pressionado.

Existem dois tipos de intents:

Intent Explícito: Especifica qual a aplicação que receberá o *Intent*, passando o *package name* ou FQCCN (Ex.: br.com.empresa.app, nome completo até chegar na classe que deseja enviar o *intent*). É executado geralmente para iniciar uma nova activity.

Intent Ímplicito: Não especifica o nome do componente, apenas declara a ação requerida. Ex.: Tenho uma foto e quero compartilhá-la, a aplicação não diz com que ela irá compartilhar, o que ela faz é passar um aviso para o sistema de que está fazendo um processo de "sending" (*action*) e o Android então procura todos os que recebem aquele *Intent* com a action específica e mostra para o usuário em qual aplicação ele poderá compartilhar a foto, como Facebook, Instagram, Whatsapp.

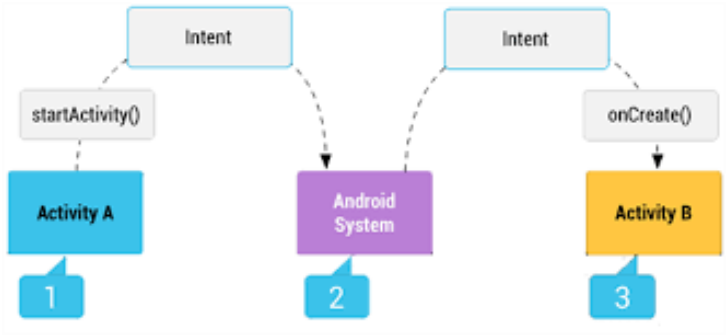


Fig. 11. Exemplo de como funciona o compartilhamento de *Intents*. O Android determina quais componentes podem atender a requisição pelo *IntentFilter*.

Intent-filter: especifica os tipos de *Intents* aos quais uma atividade, um serviço ou um broadcast receiver pode responder. A determinação pelo Android de quais componentes podem lidar com uma determinada solicitação emitida por meio de um *Intent* implícito é implementada por meio de um *IntentFilter*. Na figura acima, a Activity A cria um *Intent* descrevendo uma ação desejada e passa para startActivity(). O sistema procura em todos os apps por um *Intent filter* que pode atender à requisição de *Intent*. Quando um é encontrado, o sistema inicia a activity correspondente (Activity B) chamando o método *onCreate()* e passando a ele o *Intent*.

Com isso, foi feito uma abordagem geral sobre o funcionamento das aplicações Android. Existem algumas ferramentas que serão extremamente úteis na hora de fazer a engenharia revesera, este procedimento no Android pode ser dividido em duas partes, primeiro a **análise estática**, isto é, fazer a análise de algum código ou arquivo do aplicativo sem este estar em execução. A segunda é a **análise dinâmica**, que se difere pelo fato da aplicação estar em execução.

Ferramentas.

adb (*Android Debug Bridge*) - Ferramenta de linha de comando versátil que possibilita a comunicação e controle de um dispositivo Android conectado via USB a um computador. Com ele é possível fazer instalação de apps, depuração, listagem de pacotes, exportação e importação, entre outras ações. Funciona como cliente-servidor, possui três componentes, um cliente que envia comandos, um daemon que executa comandos no aparelho e um servidor que estabelece a comunicação entre o cliente e o servidor.

apktool – Ferramenta para engenharia reversa de binários de aplicações Android. Ele decodifica recursos para uma forma próxima da original e os reconstrói após fazer algumas modificações. Também torna mais fácil trabalhar com um aplicativo por causa do projeto, como estrutura de arquivos e automação de algumas tarefas repetitivas, como construir apk, etc.

A. Análise Estática.

Bytecode-Viewer

Jadx

Ambos decompilam código Dalvik para classes Java a partir dos arquivos presentes no pacote APK. Decodam o AndroidManifest.xml e os recursos. O jadx possui um *deobfuscator* incluso.

B. Análise Dinâmica.

Frida - Kit de ferramentas de instrumentação de código dinâmico gratuito e open source escrito em C que funciona injetando uma *engine* JavaScript no processo alvo. Permite que você execute snippets de JavaScript em aplicativos nativos em várias plataformas como Android e iOS. Implementa injeção de código escrevendo o código diretamente no processo de memória. O JavaScript é executado com acesso total à memória, funções de *hook* e até mesmo chamando funções nativas dentro do processo.

Drozer – O drozer (anteriormente Mercury) é a framework de pentest para Android. Ele possibilita escaneamento de vulnerabilidades de segurança em aplicativos e dispositivos, assumindo a função de um aplicativo e interagindo com a VM Dalvik, os endpoints IPC de outros aplicativos e o sistema operacional subjacente.

PoC de Exploração: *UnCrackable Mobile Apps – OWASP*

Nesta seção será demonstrado como realizar as análises estática e dinâmica de uma aplicação. O target escolhido para isso foi o desafio UnCrackable do OWASP.

Foi preferível usar um aparelho celular com Android para realização dos testes porque assim a experiência seria melhor aproveitada. Para instalar e utilizar o Frida para instrumentação de código, foi necessário “fazer root” no aparelho para conceder acesso de superusuário. Este procedimento não será detalhado aqui

Começando pelo UnCrackable1, temos que o objetivo proposto é burlar a verificação Anti-Root da aplicação e extrair o segredo dela.

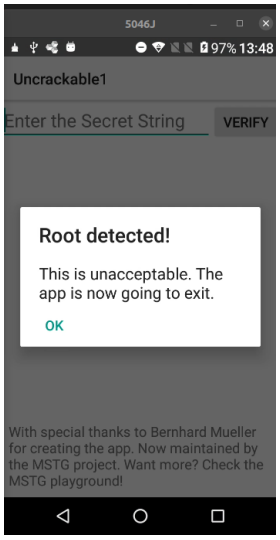


Fig.12. Aviso de detecção de acesso root concedido no aparelho, e assim o aplicativo encerra a execução.

Primeiro, será feita uma análise estática do código da aplicação para entender onde e como está sendo feita essa verificação.

Com o pacote "UnCrackable-Level1.apk" já instalado no computador e abrindo-o no jadx, após uma pequena exploração, podemos destacar a parte do código no qual é exibida a mensagem de acesso ao app concedido ou bloqueado a partir de algumas condições.

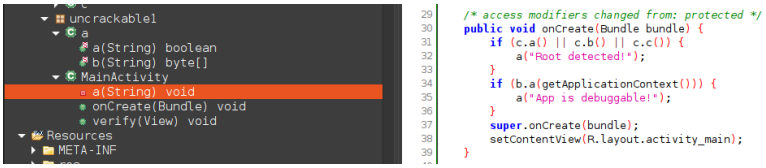


Fig.13. Parte do código que exibe a mensagem se o acesso root está concedido ou não. Classe: *sg.vantagepoint.uncrackable1.MainActivity*

Pode-se observar que se uma função das três *c.a()*, *c.b()* ou *c.c()* retornar *true*, teremos que o "*if*" será satisfeito e o alerta de root exibida. A solução proposta então para contornar essa situação será analisar os retornos de cada uma dessas funções e utilizar o Frida para alterá-las em tempo real, incitando cada uma a retornar o valor *false*.

Portanto, vamos passar brevemente por essas funções - *c.a()*, *c.b()* ou *c.c()* - para entender como funcionam.

Fig.14. Pedaço do código que nos mostra como as funções de verificação funcionam. Classe: *sg.vantagepoint.a.c*

O que se pode destacar essas verificações é que ele analisa se a palavra "su" - *c.a()* - está presente e se existem alguns arquivos relacionados ao acesso de superusuário - *c.c()* - instalados.

Agora podemos iniciar a instrumentação da aplicação. O Frida possibilita o *hook* de funções e permite que injetemos código Javascript em tempo real no aplicativo, com isso, podemos reescrever os métodos *c.a()*, *c.b()* e *c.c()*.

```
console.log("Bypass Anti-Root Uncrackable1 OWASP...");

//O Java.perform garante que o processo do Frida injetado na memoria do processo conheça o processo da aplicacao.
Java.perform(function() {

    //Modificando as funcoes que verificam o acesso root
    var c = Java.use("sg.vantagepoint.a.c");

    c.a.implementation = function() {
        console.log("Returning 'false' to c.a() call...");
        return false;
    }

    c.b.implementation = function() {
        console.log("Returning 'false' to c.b() call...");
        return false;
    }

    c.c.implementation = function() {
        console.log("Returning 'false' to c.c() call...");
        return false;
    }
});
```

Fig.15. Pedaço do script (*bypass-root.js*) que realiza o Bypass do Anti-Root.

Vale ressaltar o uso da *Java.perform()*, que garante que o processo do Frida injetado na memória do processo conheça o processo da aplicação, ou seja, recebe uma função que será executada dentro da *thread* principal em execução. O Java.use() receberá a classe onde estão os métodos.

Agora, vamos executar o frida-server no aparelho.

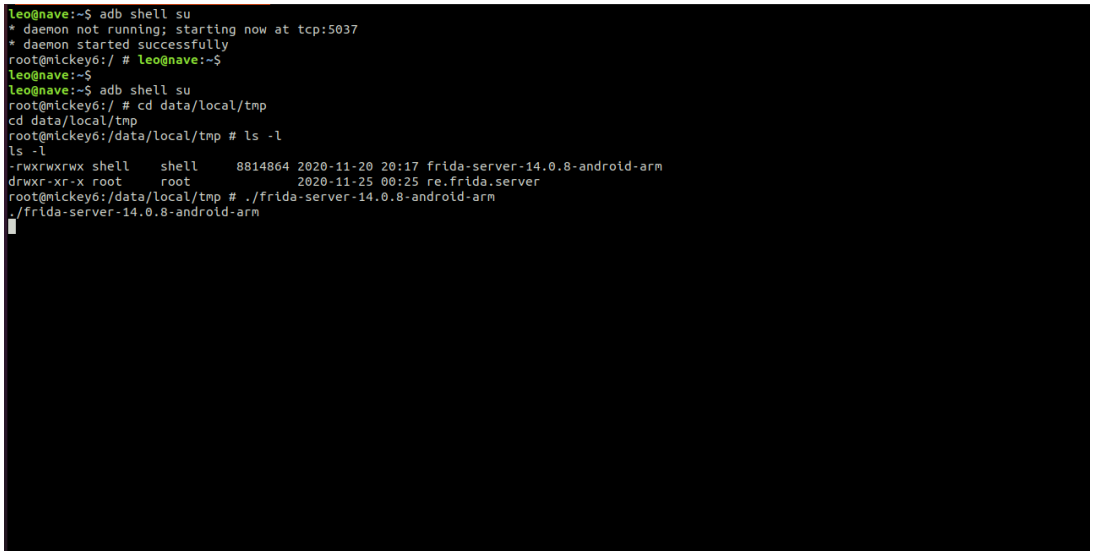


Fig.16. Rodando o *frida-server-14.0.8-android-arm*

Assim, o programa está em execução no aparelho e já é possível fazer a instrumentação. O comando *adb shell su* é para conseguir o acesso de superusuário no aparelho, e a versão do *frida-server* está ligada ao sistema do celular.

Para conferir se o procedimento foi um sucesso, podemos executar o comando *frida-ps –U* (a flag "–U" refere-se a entrada usb) para listar os processos em execução.

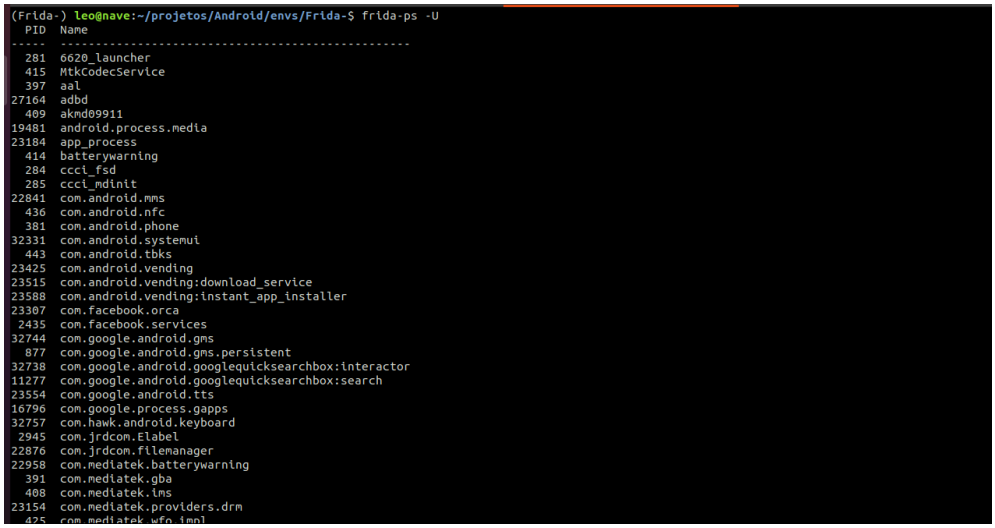


Fig.17. Com isso, vemos que o procedimento foi um sucesso.

Agora, vamos carregar esse código na aplicação, para isso, usaremos o comando: *frida -U -f owasp.mstg.uncrackable1 --no-pause -l bypass-root.js*

***-f**:* Para passar o caminho do arquivo (file).

***--no-pause**:* para que assim que rodar o código, inicie na thread principal, possibilitando o spawn automático da aplicação.

-l: Para carregar o Frida script que foi feito.

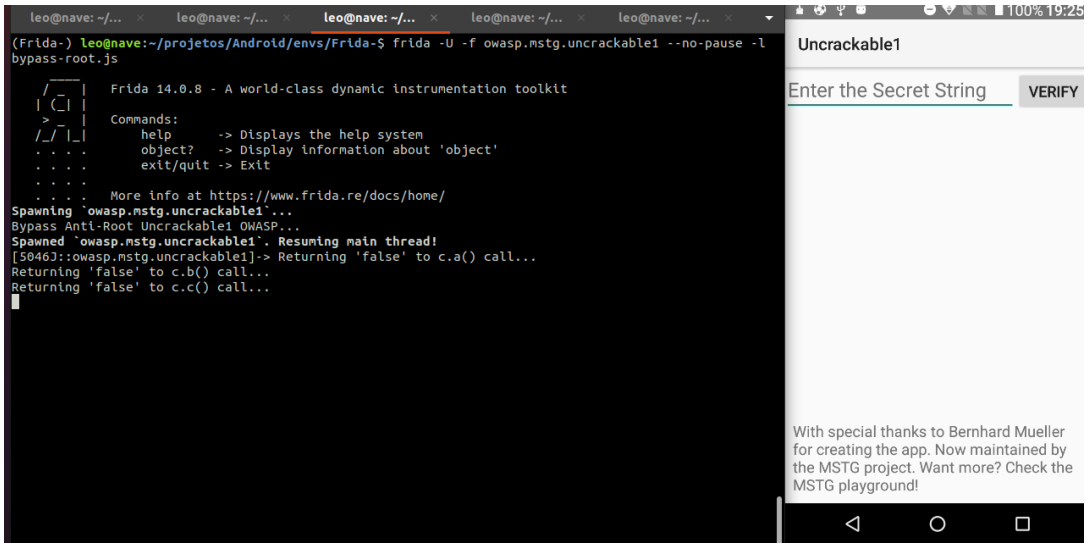


Fig.18.

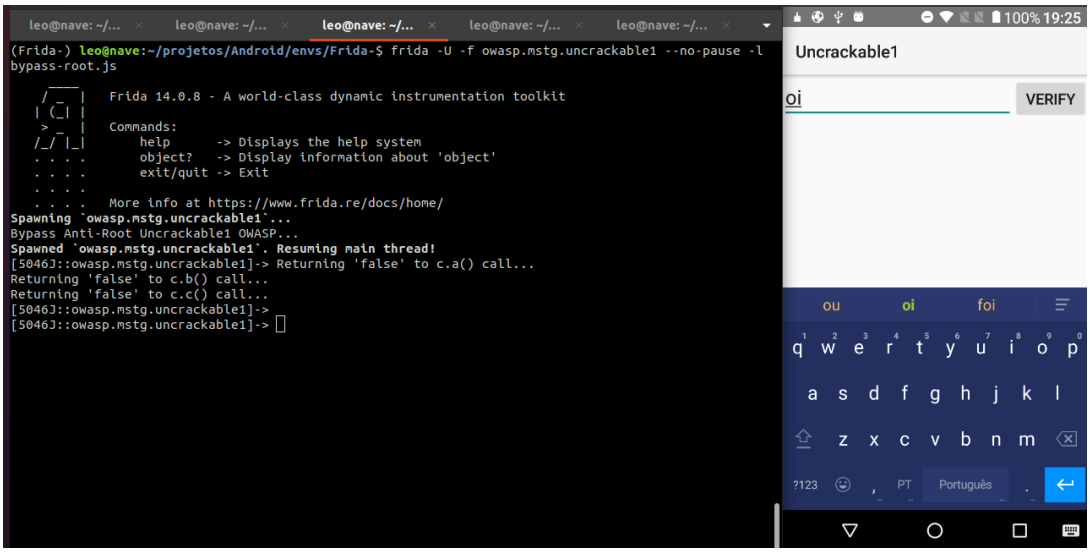


Fig.19. Percebe-se que podemos usar o aplicativo livremente após o procedimento.

A segunda parte do desafio é extrair a senha (*secret string*). Portanto, vamos voltar ao código da aplicação decompilado pelo jadx.

Fig.20. Aqui a aplicação recebe a string passada pelo usuário, e a partir do que for retornado de *a.a()*, ele exibe a mensagem de *sucesso* ou *erro.*

Fig.21.

Podemos inferir a partir da figura 20 que é feita uma comparação entre as strings do segredo e o valor que o usuário passou de entrada. Olhando mais a fundo esse bArr, vamos entender como ele é gerado.

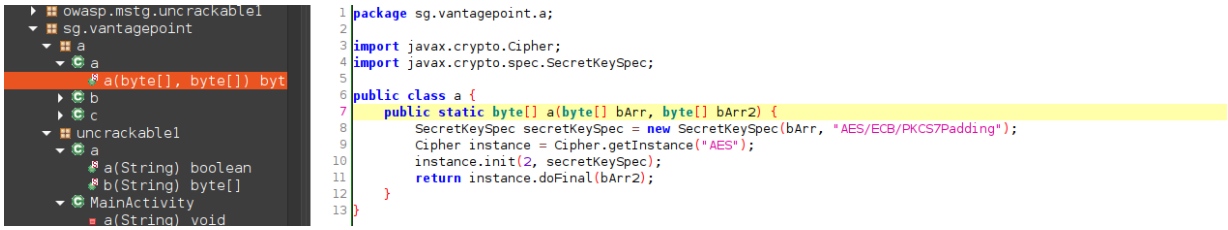


Fig.22. S*g.vantagepoint.a.a*

A string, que é comparada com a entrada, é decriptada em tempo real de execução. O algoritmo é um AES, o primeiro parâmetro de *sg.vantagepoint.a.a.a* é a chave, o segundo é o valor que será decifrado e comparado com a entrada do usuário. Com isso, *leakar* o conteúdo decifrado.

```
24    var encrypt = Java.use("sg.vantagepoint.a.a");
25
26    //Hookando a funcao dentro da classe
27    encrypt.a.implementation = function(var0,var1) {
28
29        //Chamando a propria funcao para obter o valor retornado por ela
30        var decrypt = this.a(var0,var1);
31        var secret = "";
32
33        //O valor decifrado eh um array de bytes, portanto eh necessario converter para ascii e adicionar numa string
34        for(var i = 0; i < decrypt.length; i++) {
35            secret += String.fromCharCode(decrypt[i]);
36        }
37
38        //Leakando o segredo. O decrypt sera comparado com a entrada do usuario
39        console.log("String Secret: " + secret);
40        return decrypt;
41    }
42 });
```

Fig.23. Este pedaço de código foi escrito logo abaixo do código exibido na figura 14, que já possui um Java.perform()

O *decrypt* é o segredo em bytes, o que vamos fazer é converter para ASCII e adicionar numa string para poder visualizar o valor. Será necessário retornar o *decrypt* para que ele seja comparado com a entrada do usuário.

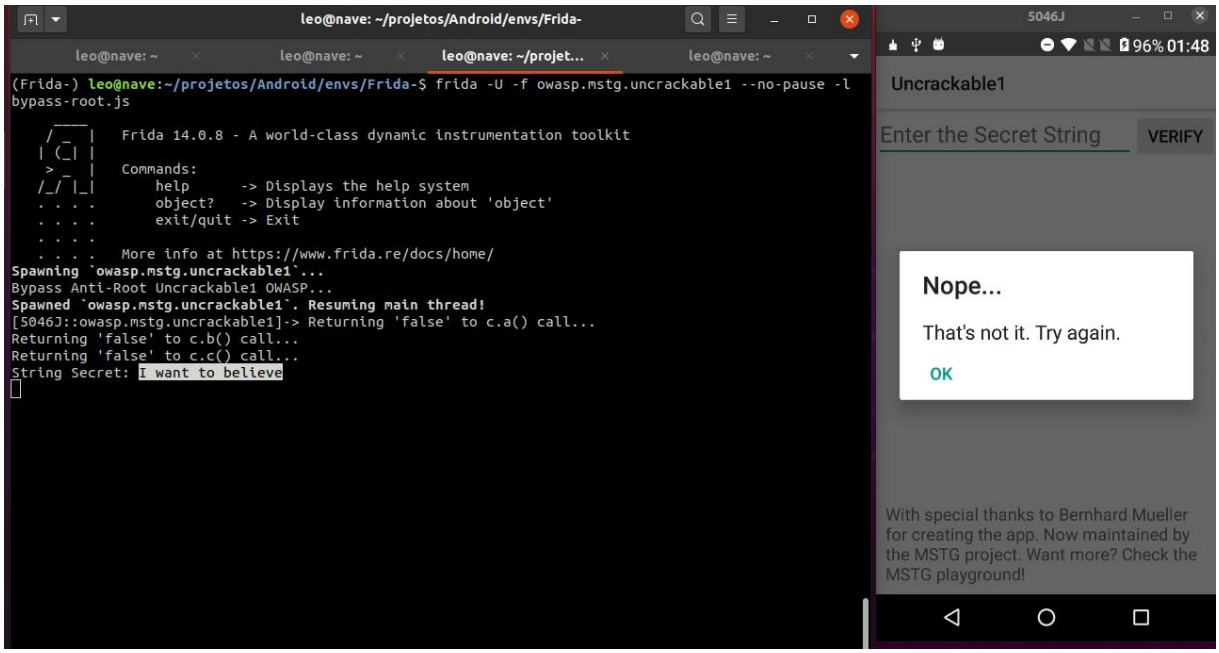


Fig.24. Quando a função hookada é executada, ela vaza o segredo.

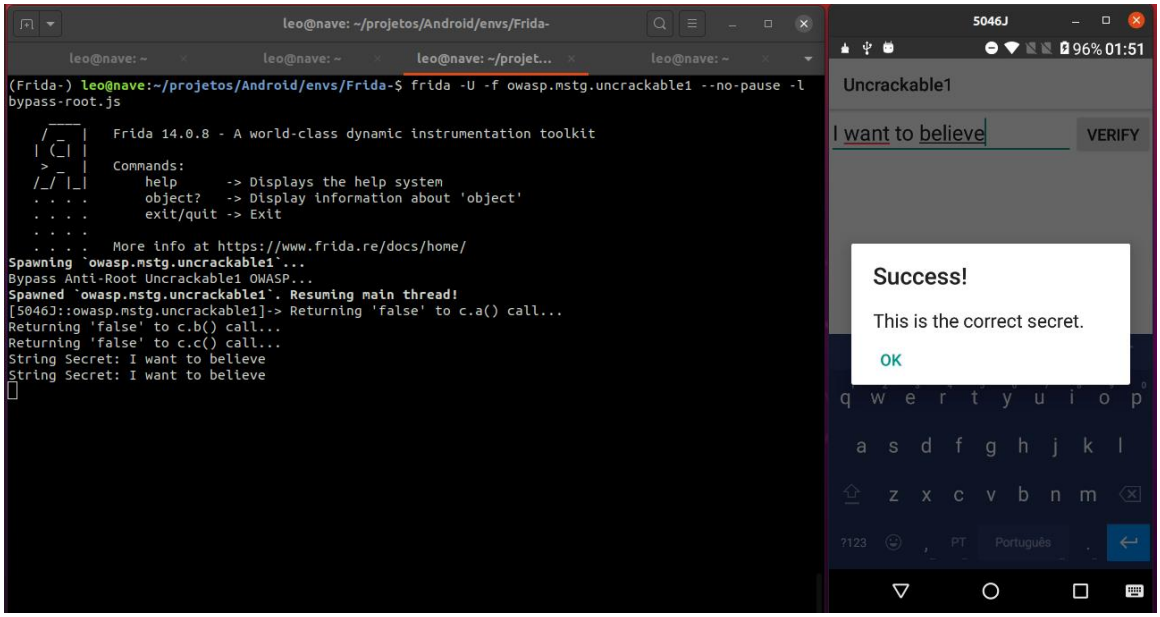


Fig.25. Desafio concluído!

Referências Bibliográficas.

Geral:

Repositório sobre *Segurança Mobile* - https://github.com/vaib25vicky/awesome-mobile-security

Playlist *Eng. Reversa de Apps Android* – MAYCON VITALI – *Hack'n Roll Academy* - https://www.youtube.com/channel/UCcYYP7JizTd24W9Mr7FIhxw

Introdução:

Android Hacker's Handbook - JOSHUA J. Drake, PAU OLIVA FORA, ZACH LANIER

Android App Reverse Engineering 101 – MADDIE STONE - https://ragingrock.com/AndroidAppRE/

Android Developer Documentation -

Fundamentos -

https://developer.android.com/guide/components/fundamentals

AndroidManifest - https://developer.android.com/guide/topics/manifest/manifest-intro

Palestra *Reverse Engineering Android Applications* - https://www.youtube.com/watch?v=m9UZnWLLurY&t=411s - TYLER LAMBERT

Ferramentas:

adb - https://developer.android.com/studio/command-line/adb

apktool - https://ibotpeaches.github.io/Apktool/

jadx - https://github.com/skylot/jadx/

Frida - https://frida.re/docs/home/

Lab + Tutoriais:

UnCrackable Mobile Apps - https://github.com/OWASP/owasp-mstg/tree/master/Crackmes

Mobile Pentesting With Frida – LAURA GARCÍA, MARTA BARRIO - https://drive.google.com/file/d/1JccmMLi6YTnyRrp_rk6vzKrUX3oXK_Yw/view

Frida Tutorial – HACK TRICKS - https://book.hacktricks.xyz/mobile-apps-pentesting/android-app-pentesting/frida-tutorial

0x02 Learning Frida by Failing – GREP HARDER - https://grepharder.github.io/blog/0x02_learning_frida_by_failing.html