Journal of King Saud University – Computer and Information Sciences 34 (2022) 1950–1957

Contents lists available at ScienceDirect

Journal of King Saud University – Computer and Information Sciences

journal homepage: www.sciencedirect.com

Android sensitive data leakage prevention with rooting detection using Java function hooking

Benfano Soewito [*], Agung Suwandaru

Computer Science Department, Binus Graduate Program – Master of Computer Science, Bina Nusantara University, Jakarta 11480, Indonesia

ARTICLE INFO

Article history:
Received 13 May 2020
Revised 1 July 2020
Accepted 14 July 2020
Available online 21 July 2020

Keywords:
Android rooting
Java hooking
Rooting detection
Rooting detection bypass

ABSTRACT

Running applications on a rooting device makes the application vulnerable to data leakage. Therefore, many applications that require a high level of security are not allowed to run on rooted device. Common technique of detecting rooted device is by using Android API to discover rooting trace. However, the detection can be bypassed using Java function hooking script by the people who want to run the app on rooted device. This research will give illustration that the bypassing process becomes more easy with automation tool and hybrid analysis. In order to create the script, we use combination of static and dynamic analysis with three phases with specific function. Phase 1 aims to detect the estimated Java method that detect rooting, phase 2 will analyze that method on an unrooted device, then phase 3 will create the bypassing script based on the previous result. We also use automation tool to speed up the static analysis. We create two types of script: one that can be used on general application, and the other one that only can be used on specific app. Those types implement different scope: one with the certain Java method, and the other one with specific parameter or return value. In the end, we find that bypassing rooting detection is not complicated if the app use Java function to detect the rooted device. To complicate bypassing process, we encourage the developers to implement more advanced detection rooting technique.

1. Introduction

Rooting is the process of getting root access on Android. To be able to access and explore the Android system freely, and take advantage of the full functionality of the Android, users are willing to rooting their devices. Some benefits gained by users after rooting include doing full backups, removing bloatware (Shao et al., 2014), running paid applications for free (Sun et al., 2015), running applications in external memory, changing user interfaces, running background services, overclocking hardware, even installing a custom OS to get the latest features and updates.

With the development of the current rooting method, rooting can be done safely and easily. Currently some OEMs provide a way to open their bootloader (Apply for unlocking Mi devices), even Google provides an image containing a binary file *su* which can be used for rooting Google Nexus (Factory Images for Nexus and Pixel Devices).

Rooting can cause security problems on Android. Malware does not have to exploit vulnerabilities in the kernel, but can easily ask users to grant root access. Because many users do not pay attention to security warnings to grant access when installing an application, malware is easier to enter and has root access. This is exacerbated by the installation of applications from unofficial stores (Zhou and Jiang, 2012) for example in China where the average level of app store trust is 47.37 over 100 (Ng et al., 2014). 37% of malware also uses root exploit to get root access (Zhou and Jiang, 2012). If the malware has gained root access, the malware can obtain sensitive data from another application's sandbox (Bojjagani and Sastry, 2017) and method calls (Casati and Visconti, 2018), get user input, change the method call when the application runs (Sun et al., 2015). If the user unroot the device, the malware can still have root access because it has created a backdoor (Zhang et al., 2014).

Because rooted device is insecure, applications requiring high level of security (sensitive applications) should not be allowed to run devices rooting. OWASP MASVS (Mobile AppSec Verification) (Xing et al., 2014) recommends detecting and responding to rooting devices by alerting the user or closing the application.

[*] Corresponding author.
E-mail address: bsoewito@binus.edu (B. Soewito).

Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

https://doi.org/10.1016/j.jksuci.2020.07.006
1319-1578/© 2020 The Authors. Published by Elsevier B.V. on behalf of King Saud University.

Rooting checking by calling the Android API can be done by checking whether there are traces of rooting left behind. Nevertheless, users can deceive the application to bypass rooting detection. Four techniques can be used to bypass rooting detection: hooking the method call at runtime, reverse engineering, patching binary, disabling rooting temporarily (Sun et al., 2015), and debugging. Hooking method call is the most dangerous technique because it is the easiest of all technique. Previous research hook the Android API based on hybrid analysis, and reading web forum. There is no detailed and structured way of doing hybrid analysis. If the source code is obfuscated, studying the source code is even more difficult.

This research will provide another approach in hooking method call so that it gives an understanding that the hooking can be made more efficient in terms of making the script and preventing the error. Therefore, we will provide solutions that developers should apply to avoid hooking techniques with this approach. In relationship with the danger of sensitive applications running on rooted device, we will provide examples of data leakage. The scope of this research is that the hooking is applied on Java function, and inspected data leakage is on application's sandbox.

Several rooting detection techniques have been found in several studies (Sun et al., 2015; Nguyen-Vu et al., 2017; Geist et al., 2016). Static and dynamic are used to detect rooting trails in the device such as installed application, file, build tags, system properties, process, directory permission, and command shell execution.

Several techniques that can be performed to bypass rooting detection (Geist et al., 2016) are hooking method call, reverse engineering, binary patching, temporarily disable root, and debugging. The most dangerous method used for bypass rooting detection is hooking method call because the application is downloaded from the Google Play store so the user will assume the application is safe, it is easier than binary patching, it can be used for the intended target application only so other applications that require root access run normally, and it does not need to enable debug.

2. Related works

The S4URC Root Checker was created by doing static analysis on applications (Nguyen-Vu et al., 2017). The results are used to hook Java API. Native hooking is also done but with trial and error based on information found in the memory device. The RDAnalyzer was made by getting rooting detection information from the XDA Developer website (Sun et al., 2015). If rooting cannot be bypassed, the static analysis will be carried out. Comparison of rooting detection techniques and rooting detection bypass techniques on iOS has been done (Geist et al., 2016).

There are other studies related to hooking, although not related to bypassing rooting detection, which can be used as a reference. The hooking technique is used for bypassing SSL Pinning (Ramírez-López et al., 2019). The SSL pinning bypass technique is explained simply and clearly using common applications (Sierra and Ramirez, 2015). Hooking techniques are also performed on rooted devices to verify sensitive data leakage (Casati and Visconti, 2018). The hooking can also be used to study malware's behavior in the sandbox and determine the risk (Jiang et al., 2018). From the literature study we can summarize some of the drawbacks of previous rooting detection bypass as follows:

1. The Android API used for rooting checks is collected from the XDA developer site and from static analysis done manually. The static analysis is used to find the API used for rooting checking, and certain keywords (method names and variable) that might be used for rooting detection. Static analysis done manually has a high likelihood of errors and requires a lot of time especially if the application has been obfuscated.
2. The return value or parameter used on the API may increase over time. For example, a newer application may check the availability of package B and the older version do not.
3. API used for rooting detection may be increased or changed with an update on the Android API.
4. If the application performs string encryption, keywords related to rooting detection cannot be read on static analysis.

3. Research method

Rooting detection bypass in this research focuses on Android API and custom-made method in Java. These are points we make to improve previous drawbacks:

1. If the application detects rooting, the method that is currently and has been called will be displayed. we will use the detected method to avoid rooting detection and call it the target method. To prevent false positives, we will detect an API list and variable that are commonly used for rooting detection.
2. Inside the target method, there is API used for rooting detection, we call it the target API. There are two options that can be used to create scripts: one which is specific to an application (option 1) and the other is more general (option 2). Option 1 hook the target API and the child target method only while the parent target method is being run. Hooking can also be done on the parent target method itself. Option 2 hook the target API with a certain return or parameter. So, the difference between these options is the scope (Fig. 3). After that, we can update our API and variable list easily.
3. We can hook the target API to find out the encrypted parameters. We can also hook the target method or the target API as long as a method is running. With option 1, we do not need to know the encrypted variable.

We divide the stages into three phases and each will do static and dynamic analysis. This can be seen in Fig. 1.

3.1. Static analysis

Static analysis is a method of studying an application from source code or results of reverse engineering. The static analysis itself has a relatively high error rate because of the complexity and obfuscation. The application must first be installed on the device via Google Play. Once installed, the APK file will be in */data/app/*. The APK file is a ZIP containing DEX file, AndroidManifest.xml, binary, and other files. With certain tools, DEX file can be converted back into Java file. This Java file is not in the original form because there is lost information when compiling Java files (Pan and Ma, 2017). In general, we can use Java files to understand the flow of the application although the result is uncertain.

Smali/Baksmali is an assembler/disassembler of the DEX file. The Baksmali process on the DEX file will produce a Smali file which is a representation of bytecode in a more readable form than DEX file. Smali file is more difficult to read than Java file because it is in assembly-like languages but it can be changed back into the DEX file. So the focus of static analysis in this study is on the Smali file where study and modifications are made. This reverse engineering process is illustrated in Fig. 2a.

3.2. Dynamic analysis

Dynamic analysis is an analysis performed on a device when the application runs. Dynamic analysis is needed to verify and reduce the false positive from static analysis while static analysis is still needed to limit the scope of dynamic analysis (Tuan et al., 2019). In this research, dynamic analysis is performed on a rooted and

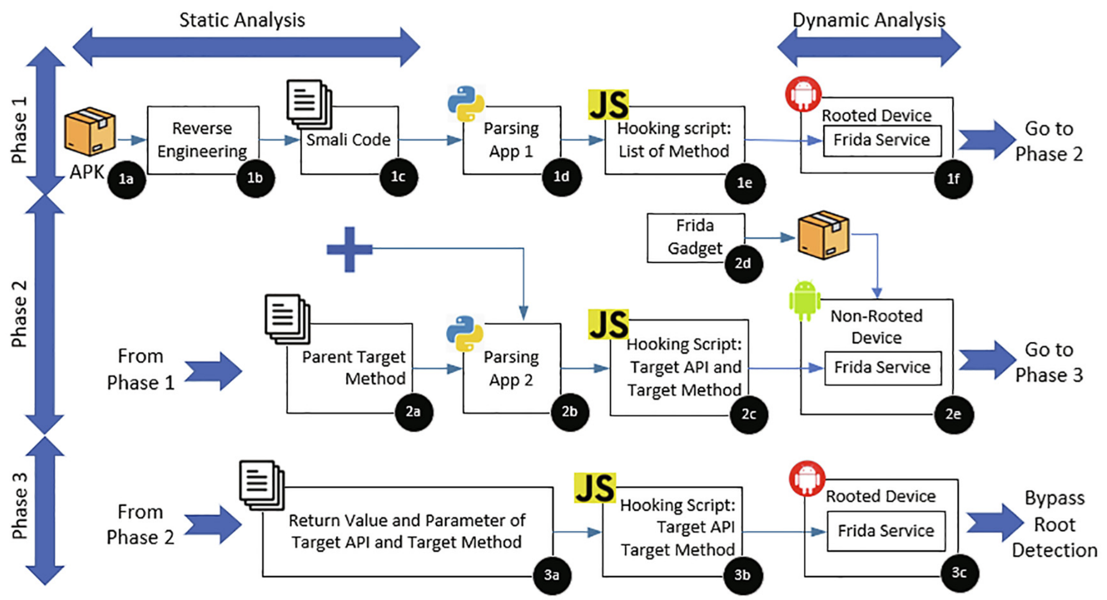


Fig. 1. Bypassing Rooting Detection Methodology using Java Function Hooking.

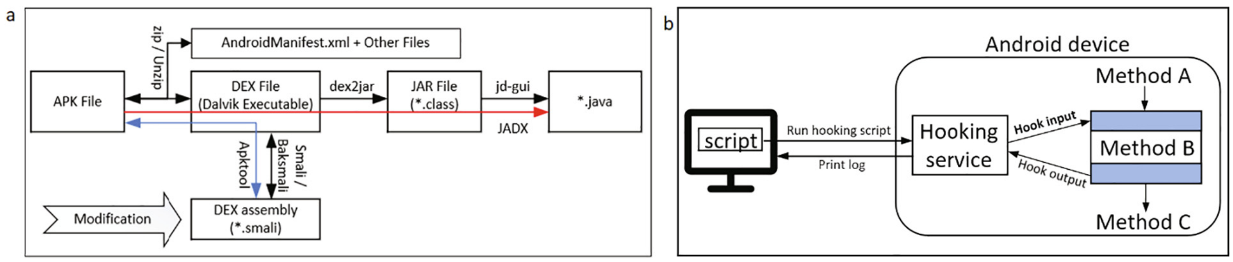


Fig. 2. (a) Static Analysis (b) Dynamic Analysis.

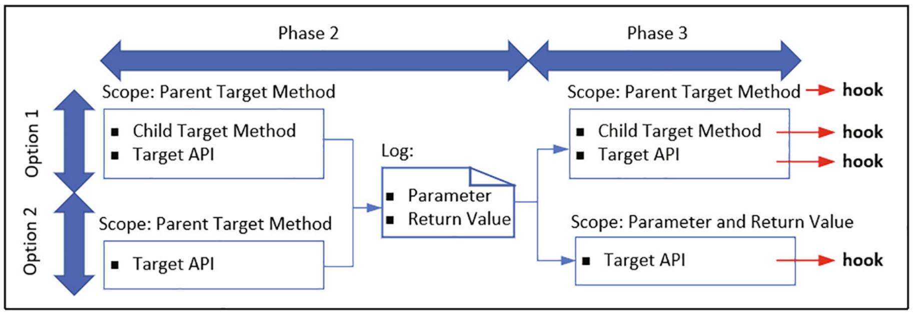


Fig. 3. Rooting Detection Bypass Option.

an unrooted device. The device will run a hooking service (Frida) and automation script (JavaScript) from the computer, then the log will be analyzed. We can hook parameter and return from a method. This is illustrated in Fig. 2b.

3.3. Phase 1

The flow of phase 1, 2, and 3 can be seen in Fig. 1. The purpose of phase 1 is to determine which method used for detecting rooting. In process 1a until 1b, once we get the APK, we can reverse engineer into the Smali file using the apktool tool. In the process 1c, we can see that each Smali file is created for each class, and folder structure shows the package or class location.

Parsing app 1d will list the method name on the Smali files marked with *.method*, class name, parameter, and return information of each method. Each method will be given an identification number. Parameter of a method from a Smali file must be correctly parsed into the script. For example parameter *I[[IILjava/lang/String;[Ljava/lang/Object;* in Smali code must be changed into *int,[[I,int, java/lang/String,[Ljava/lang/Object;* in JavaScript. In this parsing app, we include API and variable detection in method and class to reduce false positive when determining root detection method. To simplify the analysis, we can limit to only return the invoked method from certain packages or classes.

The parsing the app will form the script 1e to determine and estimate which method detects rooting. Frida service run on rooted device and the script will be run in the process 1f. Rooting is needed because Frida needs to hook the target application's ptrace and the rooting method on the log will be marked when application detect rooting. We will call this method the parent target method. Every invoked method will be logged so using string encryption (Dong et al., 2018) to hide rooting detection method is useless.

3.4. Phase 2

Fig. 3 explains two rooting detection bypass options that will be applied in phase 2 and phase 3. The purpose of phase 2 is to get the return or parameter of the target method or target API on an unrooted device. The result from phase 1 and the original Smali file will be the input of parsing app 2 to extract target API and child target method. Target API and child target method is the method / API called inside the parent target method scope in Smali file. Parsing app 2 will generate script that will be used on an unrooted device. To reduce unnecessary information in log, we can exclude some API such as logging, exception, etc. The generated script will differ depending on the option we choose in Fig. 3. On option 1, script will get the return and parameters of the target method (parent and child) and target API. On option 2, script will get return and parameters of the target API. Both options are hooked while parent target method is being run.

To run script on an unrooted device, we need to tamper with APK by inserting Frida Gadget binary (stage 2d) so that the binary will be called the first time application is run. After the tampered APK is installed, Frida service can be called with script injected from stage 2c.

3.5. Phase 3

The purpose of phase 3 is to replicate the return or parameter of target API or target method from phase 2 to bypass rooting detection. The script 3b can be made from the script 2c and phase 2 result. Option 1 and option 2 scripts will differ because the scope is different. Option 1 scope is the parent target method, while option scope 2 is the return or parameter of the API target obtained from phase 2. This is illustrated in Fig. 3 the script will be run on rooted device to make sure rooting detection can be bypassed.

3.6. Sensitive data

If the application is running on a rooted device, the application's sandbox can be exposed by malware that has root privilege. The impact will be bigger if the data is sensitive and not encrypted. Malware that has root access can also access sensitive data through the application's API call. In this study, the sensitive data observed was only in the application sandbox. Sensitive data stored insecurely is explained in the OWASP Top 10 M2 (M2: Insecure Data Storage) and OWASP MSTG Data Storage on Android (Data Storage on Android).

4. Result and discussion

In this research, there are two options that can be used. The script in phase 1 can be used in options 1 and 2 because the script is only used to determine parent target method. While in phase 2 and phase 3, each option will specify a different script. Following are examples of scripts used in phase 2 and phase 3.

Fig. 4 is phase 2 option 1 sample script where variable *a* is used to limit the logging parameter and return of *contains* API, and the return of *childTargetMethod* only when *parentTargetMethod* is being run.

Fig. 5 is phase 2 option 2 sample script where logging parameter and return of *contains* API is executed only when *parentTargetMethod* is being run. Option 2 only focuses on the target API.

Fig. 6 is phase 3 option 1 sample script that hooks the return of *contains* API and *childTargetMethod* only when *parentTargetMethod* is being run.

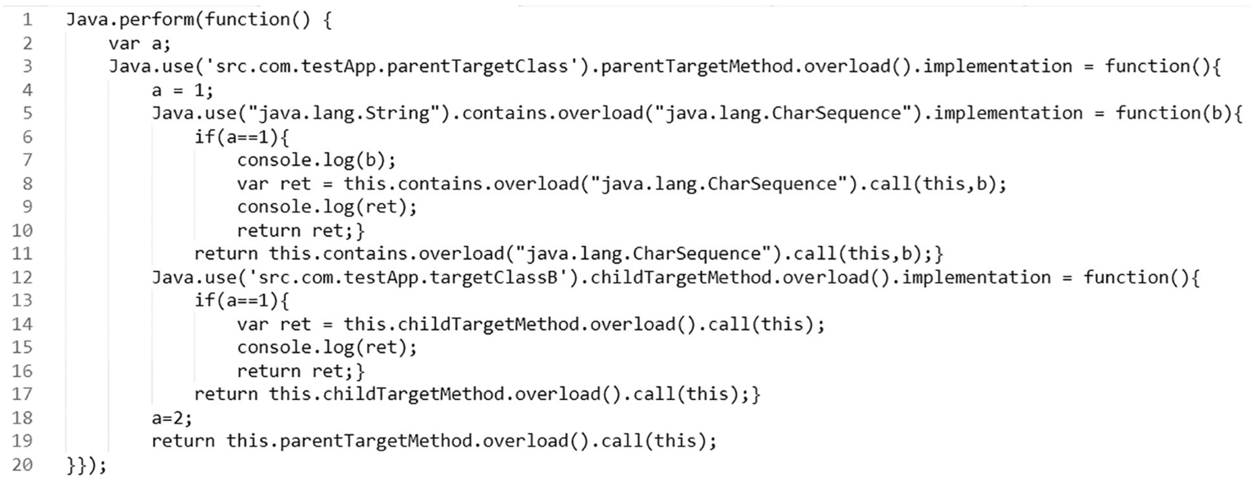
Fig. 7 is phase 3 option 2 sample script that will return *false* if parameter of *exists* API is *test-keys*. The *test-key* parameter is obtained from phase 2.

We will conduct this experiment on 4 applications. 1 Application testing will be made to see the effectiveness of rooting detection bypass on applications that have done string encryption. Three financial applications from Google Play Store are used to try to avoid rooting detection using option 1, option 2, and hybrid options.

4.1. Application with string encryption

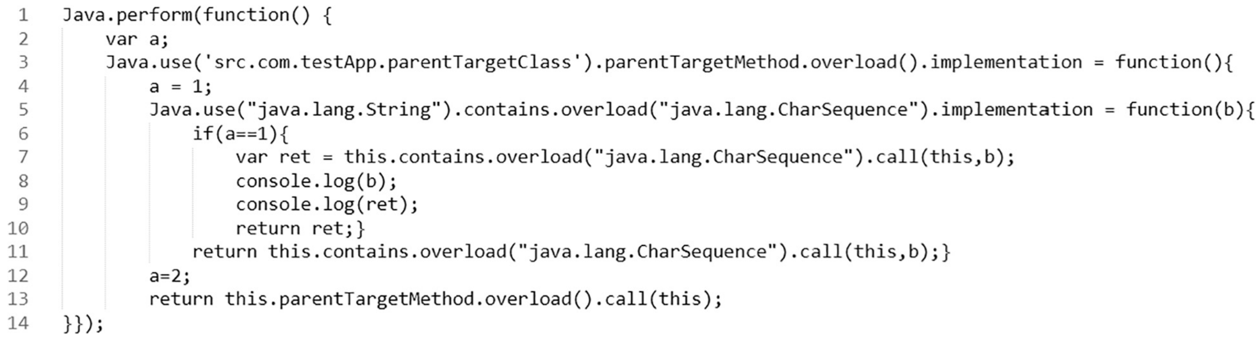
In this experiment we create application that uses string encryption to hide rooting detection variable. We implement method numbering to identify invoked method because many methods have the same name even in one class due to obfuscation.

Fig. 8.a is snippet of string encryption in Smali code that detects rooting. By scoping parent target method, we can intercept the decrypted value from parameter of target API in phase 2, then we can bypass rooting detection by changing the parameter of that



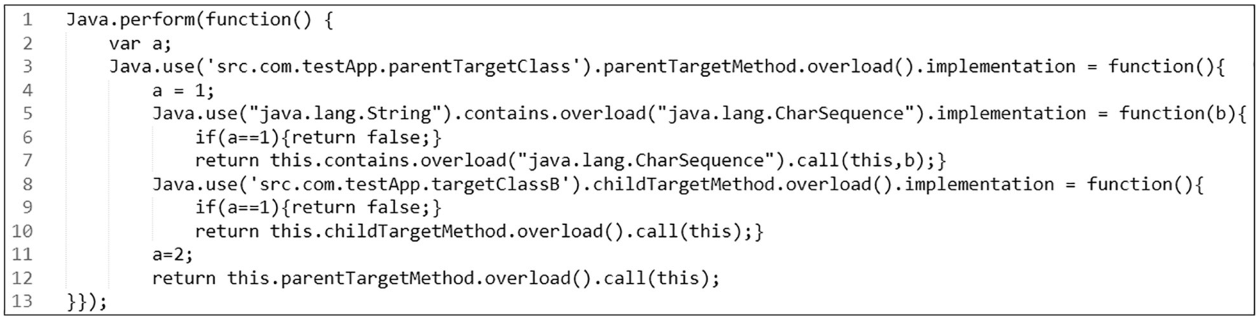
```
Java.perform(function() {
    var a;
    Java.use('src.com.testApp.parentTargetClass').parentTargetMethod.overload().implementation = function(){
        a = 1;
        Java.use("java.lang.String").contains.overload("java.lang.CharSequence").implementation = function(b){
            if(a==1){
                console.log(b);
                var ret = this.contains.overload("java.lang.CharSequence").call(this,b);
                console.log(ret);
                return ret;}
            return this.contains.overload("java.lang.CharSequence").call(this,b);}
        Java.use('src.com.testApp.targetClassB').childTargetMethod.overload().implementation = function(){
            if(a==1){
                var ret = this.childTargetMethod.overload().call(this);
                console.log(ret);
                return ret;}
            return this.childTargetMethod.overload().call(this);}
        a=2;
        return this.parentTargetMethod.overload().call(this);
}});
```

Fig. 4. Phase 2 Option 1 Sample Script.



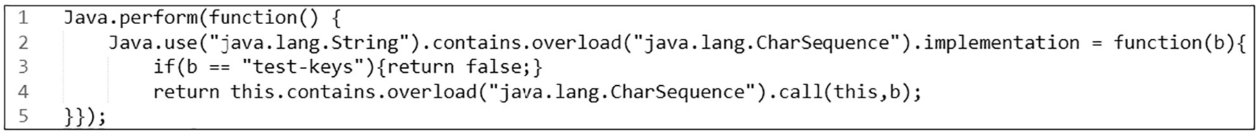
```
Java.perform(function() {
    var a;
    Java.use('src.com.testApp.parentTargetClass').parentTargetMethod.overload().implementation = function(){
        a = 1;
        Java.use("java.lang.String").contains.overload("java.lang.CharSequence").implementation = function(b){
            if(a==1){
                var ret = this.contains.overload("java.lang.CharSequence").call(this,b);
                console.log(b);
                console.log(ret);
                return ret;}
            return this.contains.overload("java.lang.CharSequence").call(this,b);}
        a=2;
        return this.parentTargetMethod.overload().call(this);
}});
```

Fig. 5. Phase 2 Option 2 Sample Script.



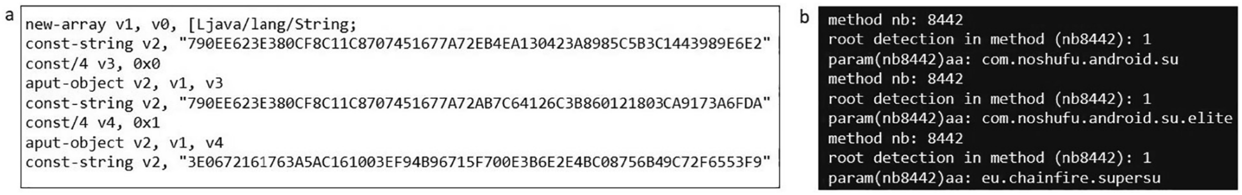
```
Java.perform(function() {
    var a;
    Java.use('src.com.testApp.parentTargetClass').parentTargetMethod.overload().implementation = function(){
        a = 1;
        Java.use("java.lang.String").contains.overload("java.lang.CharSequence").implementation = function(b){
            if(a==1){return false;}
            return this.contains.overload("java.lang.CharSequence").call(this,b);}
        Java.use('src.com.testApp.targetClassB').childTargetMethod.overload().implementation = function(){
            if(a==1){return false;}
            return this.childTargetMethod.overload().call(this);}
        a=2;
        return this.parentTargetMethod.overload().call(this);
}});
```

Fig. 6. Phase 3 Option 1 Sample Script.



```
Java.perform(function() {
    Java.use("java.lang.String").contains.overload("java.lang.CharSequence").implementation = function(b){
        if(b == "test-keys"){return false;}
        return this.contains.overload("java.lang.CharSequence").call(this,b);
}});
```

Fig. 7. Phase 3 Option 2 Sample Script.



```
new-array v1, v0, [Ljava/lang/String;
const-string v2, "790EE623E380CF8C11C8707451677A72EB4EA130423A8985C5B3C1443989E6E2"
const/4 v3, 0x0
aput-object v2, v1, v3
const-string v2, "790EE623E380CF8C11C8707451677A72AB7C64126C3B860121803CA9173A6FDA"
const/4 v4, 0x1
aput-object v2, v1, v4
const-string v2, "3E0672161763A5AC161003EF94B96715F700E3B6E2E4BC08756B49C72F6553F9"
```

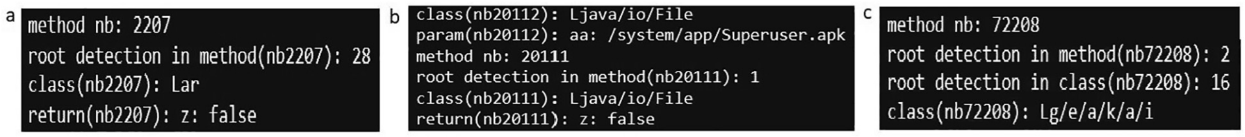
```
method nb: 8442
root detection in method (nb8442): 1
param(nb8442)aa: com.noshufu.android.su
method nb: 8442
root detection in method (nb8442): 1
param(nb8442)aa: com.noshufu.android.su.elite
method nb: 8442
root detection in method (nb8442): 1
param(nb8442)aa: eu.chainfire.supersu
```

Fig. 8. (a) String Encryption in Smali File (b) Decrypted String Detecting Rooting Found.

API once the decrypted parameter if hooked. Fig. 8.b shows the decrypted value of variable in Fig. 8.a.

4.2. Implementation with option 1

We use different financial application from Google Play for implementation with option 1, option 2, and hybrid. After obtaining the parent target method in phase 1, we get the return of child target method in phase 2. Fig. 9.a shows that return method 2207 (child target method) is *false* on rooted device. In method 2207, we also detect 28 method variables and APIs that indicate that this method detects rooting. If *false* return value of that method is applied in phase 3 when parent target method is being run, rooting detection can be bypassed.



```
method nb: 2207
root detection in method(nb2207): 28
class(nb2207): Lar
return(nb2207): z: false
```

```
class(nb20112): Ljava/io/File
param(nb20112): aa: /system/app/Superuser.apk
method nb: 20111
root detection in method(nb20111): 1
class(nb20111): Ljava/io/File
return(nb20111): z: false
```

```
method nb: 72208
root detection in method(nb72208): 2
root detection in class(nb72208): 16
class(nb72208): Lg/e/a/k/a/i
```

Fig. 9. (a) Analyzing Child Target Method for Option 1 (b) Analyzing Target API for Option 2 (c) Rooting Detection in Class for Hybrid Option.

4.3. Implementation with option 2

Fig. 9.b is log from phase 2 where target API is being analysed to get parameter and return on unrooted device. Root detection check in method equals to 1 because the app check API that might detect rooting in parent target method, and in this case variable is not detected because it is located in class variable. That log shows that target API 20112 and 202111 is connected. So, to bypass rooting detection, we must always return API 20111 *false* when API 20112 have certain parameters from phase 2 e.g. */system/app/Superuser.apk*.

4.4. Implementation with hybrid option

Fig. 9.c is log from phase 2 that shows root detection variable in the class. In this experiment, several methods detect rooting so we apply multiple parent target methods. We can implement different option for each parent target method. These are the techniques we use to bypass rooting detection on multiple parent target methods:

- Hooking return value of the parent target method (option 1).
- Hooking target API when the parent target method is being run (option 1).
- Hooking child target method when the parent target method is being run (option 1).
- Hooking target API so that always return constant value when certain parameters are found (option 2).

4.5. Data sensitive

In this research, the sensitive data in the sensitive application's sandbox is examined which can only be accessed in rooted device. We found that sensitive data in the form of plaintext such as session, log, XML, SQLite database, or other files. We also found photo of ID card, and sensitive documents. Therefore, more advanced technique is needed to detect rooting that cannot be bypassed so that sensitive data especially on the sandbox cannot be found.

4.6. Discussion

Running applications on a rooted device makes the application vulnerable to data leakage. Any data movement in the application can be stolen by hooking up the method call. Data in the sandbox can also be stolen. Application's rooting detection can be bypassed by hooking Java function. Obfuscation such as identifier renaming and string encryption cannot prevent rooting detection bypass. In this research, firstly we determine which method detecting rooting, then we observe the method on an unrooted device, finally we create bypassing script.

We use two options Java function hooking. Option 1 is the easiest option because we can hook the API or custom method then limit hooking when a parent method is being run. We can hook parameter or return without having to know the original value. The drawback of this option is that we cannot use the same script for other applications or for the same application with different version. This is because the method name can change due to obfuscation or developer use another name. Errors can occur if the parent target method is used besides rooting detection so the right scope is needed.

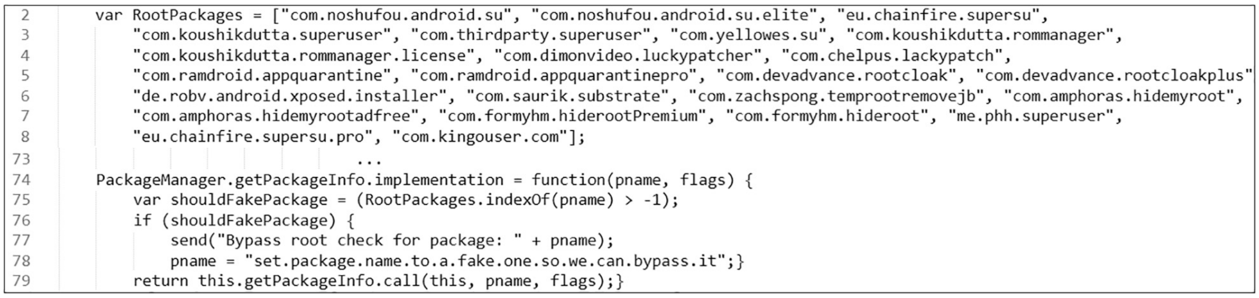
Option 2 script is the most difficult to create because we need to know the parameter or return of target API. Sometimes, some APIs are correlated in Smali file so we to hook the right API with the right value and reading API documentation necessary. Another difficulty is that if a parent target method has child target methods and target APIs, we sometimes have to analyze the flow of the application. The advantage of option 2 is that the script could be made for other application if we update the API and needed variable in the script.

In the previous studies, rooting detection was done by checking the APIs and variables in the source code (static analysis), then the results were used to bypass rooting detection by hooking Java function. There are no detailed steps on how static and dynamic analysis are carried out, and the source used is not provided. Therefore, we will compare the rooting detection bypass technique found in the open source code (fridantiroot) with the technique we use. We assume that that source code is similar to the one found on previous method because the script was made general for many applications.

Fig. 10 is script snippet from Frida codeshare. It is seen certain parameters become the scope of target API. This means that the technique used in that script is the same as the technique we use in this research option 2. To increase the efficiency of rooting detection, we detect invoked methods when the application detects rooting in phase 1.

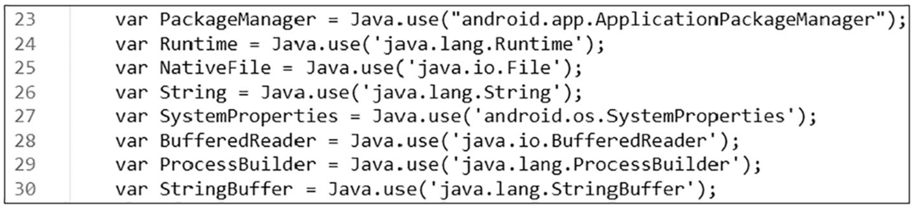
Fig. 11 is the same script from Frida codeshare. It appears that the code hooks several APIs. To add list of APIs, firstly we analyse the parent target method in phase 2. To decrease false positive in determining parent target method, we detect variables in the method (Fig. 9.a) and in the class (Fig. 9.c) that may be used for rooting detection. Detection of variables is done in phase 1, while the use of that variable on the API is done in phase 2. By knowing which APIs are used, we can update list of APIs in Fig. 11. At the same time, we also update the list of variables then apply them to parsing app 1. If the variables that detect rooting are encrypted, these variables can be detected in phase 2. Updating list of variables and APIs lead to increase of bypassing rooting detection efficiency.

Option 1 is not found in the general rooting detection bypass technique as in Fig. 10 and in previous studies because option 1 is specific to a particular application. To prevent error when the application is run, we only run child target method and target API only when parent target method is being run. So, if child target



```
var RootPackages = ["com.noshufou.android.su", "com.noshufou.android.su.elite", "eu.chainfire.supersu",
    "com.koushikdutta.superuser", "com.thirdparty.superuser", "com.yellowes.su", "com.koushikdutta.rommanager",
    "com.koushikdutta.rommanager.license", "com.dimonvideo.luckypatcher", "com.chelpus.lackypatch",
    "com.ramdroid.appquarantine", "com.ramdroid.appquarantinepro", "com.devadvance.rootcloak", "com.devadvance.rootcloakplus"
    "de.robv.android.xposed.installer", "com.saurik.substrate", "com.zachspong.temprootremovejb", "com.amphoras.hidemyroot",
    "com.amphoras.hidemyrootadfree", "com.formyhm.hiderootPremium", "com.formyhm.hideroot", "me.phh.superuser",
    "eu.chainfire.supersu.pro", "com.kingouser.com"];
                          ...
PackageManager.getPackageInfo.implementation = function(pname, flags) {
    var shouldFakePackage = (RootPackages.indexOf(pname) > -1);
    if (shouldFakePackage) {
        send("Bypass root check for package: " + pname);
        pname = "set.package.name.to.a.fake.one.so.we.can.bypass.it";}
    return this.getPackageInfo.call(this, pname, flags);}
```

Fig. 10. Detection Rooting Bypass Script from Frida Codeshare (fridantiroot).



```
var PackageManager = Java.use("android.app.ApplicationPackageManager");
var Runtime = Java.use('java.lang.Runtime');
var NativeFile = Java.use('java.io.File');
var String = Java.use('java.lang.String');
var SystemProperties = Java.use('android.os.SystemProperties');
var BufferedReader = Java.use('java.io.BufferedReader');
var ProcessBuilder = Java.use('java.lang.ProcessBuilder');
var StringBuffer = Java.use('java.lang.StringBuffer');
```

Fig. 11. Target API from Frida Codeshare (fridantiroot).

method or target API is used other than for rooting detection, the script does not cause error.

Our research can be used for application owners to assess how resistant their application against rooting detection bypass, and for developer to implement more robust detection. Generally, researcher can also use it to analyze application's behavior even on unrooted device.

5. Future work

This research can be developed into the following studies:

- This research's methodology can be modified and used to bypass jailbreak detection on iOS devices then measure its effectiveness. The possibility of using automation can also be considered. Previous research (Kellner et al., 2019) and scripts openly available can be used as a starting point to study jailbreak detection and evasion using the hooking function.
- On Android, rooting detection can be done by native code and script can be developed to bypass it. Technique to analyze application using hooking on binary file has been done (Totosis and Patsakis, 2018). Several binary detecting rooting can be bypassed but still using manual inspection (Nguyen-Vu et al., 2017). Future research will focus on developing bypass methodology for native code.

Final goal of these both future works are to create more advanced rooting detection difficult to bypass. Action prior to bypassing rooting detection might be taken into consideration such as deobfuscation (Kan et al., 2019).

6. Conclusion

We found that sensitive data might have been found in the application's sandbox on the rooted device because developers do not properly encrypt sensitive data. We prove that the rooting detection bypass technique with Java function can be easily done. Therefore, we need a detection technique which is difficult to be bypassed. Some techniques that developers can use to make bypassing difficult are:

1. Using native library made with C to detect rooting. This technique can actually be bypassed by hooking native code but it is more difficult to reverse engineer the binary file.
2. Applying hooking detection and obfuscation to the native library (Lim et al., 2017). This technique makes reverse library native engineering more difficult. Hooking is also more difficult because we need to bypass hooking detection first.
3. Most applications that detect rooting will pop up an alert that requires user interaction. To make determining rooting detection method difficult, the application should be left running but with limited functionality. Alert can be used that don't require user interaction.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The authors would like to thank the Ministry of Research, Technology and Higher Education of the Republic of Indonesia for their support in making this research possible. This research was supported by Awarded Research Grant scheme 2019, No. 12/AKM/PNT/2019, 039/VR.RTT/IV/2019.

References

Shao, Y., Luo, X., Qian, C., 2014. Rootguard: protecting rooted android phones. Computer 47 (6), 32–40.

Sun, S.T., Cuadros, A., Beznosov, K. Android rooting: Methods, detection, and evasion. In Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices 2015 Oct 12 (pp. 3-14).

Apply for unlocking Mi devices. Retrieved from miui: from: https://en.miui.com/unlock/.

Factory Images for Nexus and Pixel Devices. Retrieved from google: https://developers.google.com/android/images.

Zhou, Y., Jiang, X. Dissecting android malware: characterization and evolution. In 2012 IEEE Symposium on Security and Privacy 2012 May 20 (pp. 95-109). IEEE.

Ng, Y.Y., Zhou, H., Ji, Z., Luo, H., Dong Y. Which android app store can be trusted in china? In 2014 IEEE 38th Annual Computer Software and Applications Conference 2014 Jul 21 (pp. 509-518). IEEE.

Bojjagani, S., Sastry, V.N. VAPTAi: A Threat Model for Vulnerability Assessment and Penetration Testing of Android and iOS Mobile Banking Apps. 2017 IEEE 3rd International Conference on Collaboration and Internet Computing (CIC) 2017 Oct 15 (pp. 77-86). IEEE.

Casati, L., Visconti, A., 2018. The dangers of rooting: data leakage detection in android applications. Mobile Inf. Systems 2018.

Zhang, Z., Wang, Y., Jing, J., Wang, Q., Lei, L. Once root always a threat: analyzing the security threats of android permission system. In Australasian Conference on Information Security and Privacy 2014 Jul 7 (pp. 354-369). Springer, Cham.

Xing, L., Pan, X., Wang, R., Yuan, K., Wang, X. Upgrading your android, elevating my malware: privilege escalation through mobile os updating. 2014 IEEE Symposium on Security and Privacy 2014 May 18 (pp. 393-408). IEEE.

Nguyen-Vu, L., Chau, N.T., Kang, S., Jung, S., 2017. Android Rooting: An Arms Race Between Evasion and Detection. Security and Communication Networks.

Geist, D., Nigmatullin, M., Bierens, R. Jailbreak/Root Detection Evasion Study on iOS and Android. MSc System and Network Engineering. 2016 Aug 23.

Ramírez-López, F.J., Varela-Vaca, Á.J., Ropero, J., Luque, J., Carrasco, A., 2019. A framework to secure the development and auditing of SSL pinning in mobile applications: the case of android devices. Entropy 21 (12), 1136.

Sierra F, Ramirez A. Defending your android app. In Proceedings of the 4th Annual ACM Conference on Research in Information Technology 2015 Sep 29 (pp. 29-34).

Jiang, X., Liu, M., Yang, K., Liu, Y., Wang, R., 2018. A Security Sandbox Approach of Android Based on Hook Mechanism. Security and Communication Networks.

Pan, J.Y., Ma, S.H. Advertisement removal of Android applications by reverse engineering. In 2017 International Conference on Computing, Networking and Communications (ICNC) 2017 Jan 26 (pp. 695-700). IEEE.

Tuan, L.H., Cam, N.T., Pham, V.H., 2019. Enhancing the accuracy of static analysis for detecting sensitive data leakage in Android by using dynamic analysis. Cluster Comput. 22 (1), 1079–1085.

Dong, S., Li, M., Diao, W., Liu, X., Liu, J., Li, Z., et al., 2018. Understanding Android obfuscation techniques: a large-scale investigation in the wild. In: International Conference on Security and Privacy in Communication Systems. Springer, Cham, pp. 172–192.

M2: Insecure Data Storage. Retrieved from OWASP: https://owasp.org/www-

project-mobile-top-10/2016-risks/m2-insecure-data-storage.
Data Storage on Android. Retrieved from github: https://github.com/OWASP/owasp-mstg/blob/master/Document/0x05d-Testing-Data-Storage.md.
fridantiroot. Retrieved from Frida: https://codeshare.frida.re/@dzonerzy/fridantiroot/.
Kellner, A., Horlboge, M., Rieck, K., Wressnegger, C. False sense of security: a study on the effectivity of jailbreak detection in banking apps. In 2019 IEEE European Symposium on Security and Privacy (EuroS&P) 2019 Jun 17 (pp. 1-14). IEEE.
Totosis, N., Patsakis, C. Android hooking revisited. In 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech) 2018 Aug 12 (pp. 552-559). IEEE.
Kan, Z., Wang, H., Wu, L., Guo, Y., Xu, G. Deobfuscating Android native binary code. In 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion) 2019 May 25 (pp. 322-323). IEEE.
Lim, K., Jeong, J., Cho, S.J., Choi, J., Park, M., Han, S., Jhang, S. An anti-reverse engineering technique using native code and obfuscator-LLVM for Android applications. In Proceedings of the International Conference on Research in Adaptive and Convergent Systems 2017 Sep 20 (pp. 217-221).