DAMN - A Debugging and Manipulation Tool for Android Applications

Gerald Schoiber, MSc.
schoiber@ins.jku.at

Univ.-Prof. Priv.-Doz. DI Dr. Rene Mayrhofer
mayrhofer@ins.jku.at

Michael Hölzl, MSc.
hoelzl@ins.jku.at

Institute of Networks and Security
Altenbergerstr. 69
Linz, Austria 4040

ABSTRACT

Mobile developers tend to use source code obfuscation to protect their code against reverse engineering. Unfortunately, some developers rely on the idea that obfuscated applications also provide additional security. But that is not the case since mistakes in design are still present and can be used for arbitrary attacks. However, manually analyzing such obfuscated applications is time consuming for researchers due to the complexity of the generated code.

Our *d*ebugging *a*nd *man*ipulation tool (*DAMN*) offers a new way of investigating Android applications, including obfuscated ones. It combines static source code reversing with dynamic manipulation techniques to get rid of obfuscation penalties and supports the investigator during the analyzing process. DAMN can display the reversed source code, pause any application at any given time and allows to manipulate its state. All those features make DAMN a powerful reversing and analyzing tool for manual investigations of obfuscated Android applications.

Keywords

Android, Reverse Engineering, Code Analysis, Debugging

1. INTRODUCTION

Automated analysis tools play an important role when it comes to application analysis as they can handle huge amount of applications in short time. This is relevant if we take a look at the PlayStore where a lot of applications are available. However, those automated tools are not perfect and some malicious applications bypass the audit. On the other hand, manual analysis of each application would not be sufficient because it is too time consuming and is not a guarantee to find every malicious behavior. But manual analysis are important if new concepts of malicious applications are on the market as they can only be hardly detected by automated analysis tools. There are many ways of manually investigating applications as for example network traffic analysis or reverse engineering (Section 4.1). Especially reversing applications can help to find possible vulnerabilities and is therefore heavily used. But this reversing tools do not or only barely give support for obfuscated code. Obfuscation makes it serve to manually analyze code because it splits packages, classes and even methods in smaller parts as well as rename them into meaningless names. As source code obfuscation became pretty standard nowadays, it can slowdown investigations and make it harder to find possible weaknesses. In contrast to static analyzing techniques, dynamic analyzing does not struggle with obfuscated source code because the applications still have to be executable and have to behave the same way as none obfuscated ones. Our debugging and manipulation tool DAMN was build with obfuscation in mind and combines static with dynamic analyze techniques to support the manual analyzing process. Moreover, it is possible to debug executed applications without having the source code available. DAMN is also capable of manipulation on executed applications which supports the investigator by analyzing the behavior under different conditions. This makes testing against several possible vulnerabilities easy and fast.

MoMM '16, November 28 - 30, 2016, Singapore, Singapore

ISBN 978-1-4503-4806-5/16/11...$15.00

DOI: http://dx.doi.org/10.1145/3007120.3007161

2. RELATED WORK

Because DAMN is an analyzing tool in the very first place, we took a look at Android analyzing tools. Most of them are full automated and have no support of manual investigations. The following list is a subset of available tools and do not have the intend to be complete.

TaintDroid. Enck et al. [6] created TaintDroid which tracks the flow of privacy sensitive data through applications in realtime. They label/tag such sensitive data with a taint. After that, they can trace the data whenever an application will access it. In such a case, TaintDroid notifies the user (or other applications) about that. Another study they published in the paper uncovers that about two-thirds of thirty popular Android applications leak sensitive data. Unfortunately TaintDroid have no implementation to block those unwanted data leaks automatically or manually.

AppFence. Another tool is called AppFence which was written by Hornyack et al. [8]. It is using TaintDroids tainting approach for its implementation which is also protecting the user against sensitive data leaks. They provide two approaches for protection. The first one fakes data if an application is requesting sensitive data (e.g. location data). This

solution is also known as shadowing. The second attempt is to prevent exfiltration of data. Whenever AppFence is detecting a tainted data will be written to a socket, it will drop the data immediately and either send a fake conformation or lets the application believe that the device is in airplane mode.

DroidScope. In differ to the tools above, DroidScope, which was written by Yan and Yin [18], does not run on real devices. Instead, it is running on the Android emulator which makes it scalable. The focus rely on malware detection rather than on manipulation.

DroidTrace. The only implementation which provides forward execution is DroidTrace developed by Zheng et al. [19]. It uses *ptrace* (process trace [11]) and disassembling techniques for its dynamic analysis. The basic approach is to disassemble an application into *smali-code*[1], alter and repack it afterwards. After that, they are able to trigger dynamic load behaviors on the manipulated applications for its analyzing purpose.

API Monitor & Aurasium. Xu et al. [17] wrote a tool named API Monitor & Aurasium. It also uses repacking to add additional code into the application with the *apktool*[2]. If an violation occurs, the user gets notified and can allow or deny it.

Mobile-Sandbox. Another automatic test suit solution is Mobile-Sandbox. It combines static as well as dynamic analyzing techniques for testing. Furthermore, it logs native API calls [13].

ANANAS. Another automated static and dynamic malware analysis framework is called ANANAS. It is build to be expendable and allows to extend functionality by writing plugins. Those plugins can react to events which are raised by the framework and execute additional code. Furthermore, logs are stored into a database for filtered reports afterwards. ANANAS also uses the Android emulator for execution which can simulate user interactions over a scripting language [4].

ANDRUBIS. In contrast to the previous solution, ANDRUBIS is using *QEMU* [1] as its runtime environment [15], which is a generic open source machine emulator and virtualizer. It is also an automated static and dynamic malware analysis tool which analyzes the manifest file and the byte code. In addition, it can analyze network traffic to complete the investigation. ANDRUBIS analyzed one million applications from the *PlayStore* and published the results in this paper [9].

Google Bouncer. Google also tests applications which are in the store. Their automated dynamic analyzing tool is called Google Bouncer and less is known about it because Google keeps it secret. *Jon Oberheide* and *Charlie Miller* were trying to get some more information and published the result [10]. If malicious behavior is detected, it will remove the application from the store. *Nichoas Percocos* showed at the *Black Hat*[3] back in 2012 that Bouncer is not a perfect analysis tool and that it is possible to leverage it [12].

To sum up, the currently available tools are mostly automated analyzing tools. They are using both, static and dynamic analyzing techniques. Because static analyzing can hardly detect dynamic code execution and struggles with the completeness of the reversed source code it needs the dynamic analyzing part to overcome this issue. The downside of dynamic analysis is that they are very complex and therefore it is not always feasible to analyze all possibilities. Because of that it is necessary to perform additional manual code analysis.

[1]https://github.com/JesusFreke/smali
[2]http://ibotpeaches.github.io/Apktool/
[3]https://www.blackhat.com/

3. CONCEPT

Idea. DAMN combines static and dynamic analysis methods to support the investigator during the manual investigation of Android applications. The static part will reverse the application and extract the Java source code (Section 4.1). As this source code can be obfuscated we need another way to overcome the problem with obfuscated reversed source code. DAMN uses the dynamic component to support the investigator finding relevant source code sections in less time. The application which should be analyzed can be launched on the Android device as usual. Then the investigator can navigate through the app and if an interesting part appears on the screen, for example a login, it can be paused. At this point, DAMN will show the method which got called. Further calls can also be investigated by simply single step through the application.

Features. To make the investigation more effortless, DAMN provides a browser interface for a second screen to show the source code and to control the app. Further, it is possible to investigate the parameters and the return values of the called methods. This helps to understand what the obfuscated method is supposed to do. As an additional information, the investigator can also inspect field values of the actual class. To test against different scenarios, DAMN provides the functionality to change the values of those parameters, return values and fields directly in the browser. This can be used to get feedback how the application will behave on different conditions. This can be used to quickly check against possible flaws and vulnerabilities.

4. DAMN

DAMN consists of multiple components which are playing closely together. The main components are explained in more detail in below.

4.1 Reverse Engineering

One feature of DAMN is to reverse engineer Android applications. The resulting Java source code is the base of manual static analyzing [3]. While reverse engineering is a widespread term for various techniques and decompiling is only one method of this topic [2], we use reversing as a synonym for decompiling in this paper.

Android Decompiler. Although Android is based on Java, there are Android specific decompilers which make use of the additional information that is stored in the apk file. *Jadx*[4] is such a decompiler and the generated source code is much more cleaner and more readable as Java specific ones. Unfortunately, DAMN can not directly use a decompiler because those tools are using native libraries which are not available for the ARM architecture. To make it more effortless,

[4]https://github.com/skylot/jadx

DAMN provides a simple BASH script which decompiles an application on machine which is supported by the decompiler and puts the source code back onto the device.

4.2 User Interfaces

We use a second screen to provide the most usable way of display information and take control over an application without interfering the running app on the device. To achieve this, we are using a client-server pattern. The server part is running on the Android device and provides web pages which can be used to gather information and control the investigated application. Usually, both parties have to be connected over a WiFi network. But for some system applications which have been started before there is any network connection available, DAMN supports connecting through USB.

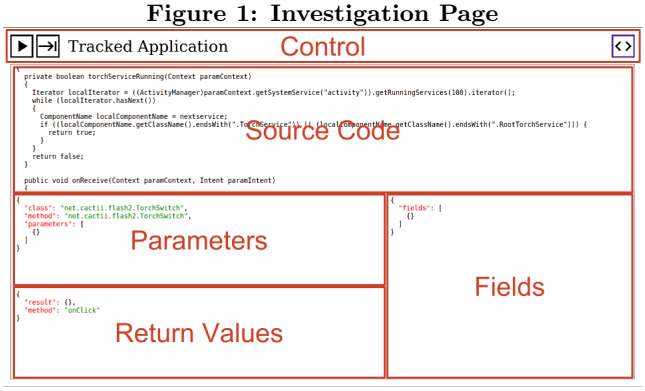
DAMN Application. The main application is a simple view pager that contains lists of installed applications and their packages. Those packages can be selected separately to help investigating only a subset instead of all. Packages are separated by a point and because we do not want to have every sub package listed, we only show the first three parts of the packages. If there are more sub packages declared, they will be selected with the leading package name. After selecting one or multiple packages, DAMN write a configuration file (Section 4.5) with those information into the installation directory of the application.

Browser Interface. A more complex interface will show up during the investigation. The second screen solution provides a comfortable way of displaying information of the running application without interfering the display of the device. As browsers are usually available on every device, we decided to use a web page to display the information. Additionally, we can control the application remotely over this interface. DAMN is using the *Civetweb* server (section 4.3) and web sockets for this purpose. The web interface is separated into two pages. The investigation page is the most notable page DAMN is displaying. It is segmented in different sections as shown in Figure 1. The *Control* section makes it possible to switch between the different runtime states (Section 4.6). This makes it possible to fully control the application at any time. In the *Source Code* section, DAMN displays the decompiled source code of an application if it is available (Section 4.1). The *Fields* section shows all declared fields and their values from the current class of the running application. Section *Parameters* and *Return Values* displaying the current executed method on call or return. Those sections will not only show the passed, respectively returned objects and values, it also allows to manipulate them directly (Section 4.7).

4.3 DAMN Server

To make it possible to display and control an investigated application at the same time, DAMN uses a customized Civetweb[5] server implementation. This component is written in native code and compiled as a standalone executable. The reason why we wrote it in native code was that it has to be executed before all Android components are started which allows to start Java based applications. This component is started on boot before other Android components are loaded [7].

Figure 1: Investigation Page



4.4 Xposed Module

One of the core components is the DAMN Xposed module which we are using to gather information about the running application as well as to have the possibility to control it. To understand how it works, we first take a closer look into Xposed.

Xposed. DAMN is using the Xposed[6] framework which is an open source project that allows to change the behavior of an application without changing the source code. All changes are done on memory which makes manipulation easy. To make this possible, Xposed loads an additional JAR file at the start of Zygote to run in its context. Zygote is the first process of all Android applications. Any app will be a fork of it and therefore have the same basic components [5]. Because of the additional JAR file Xposed can take control over the application.

Xposed provides multiple ways to interact with an application. The most remarkable ones are the hooking methods that can hook chosen methods and provide two interaction points where a developer can intercept the running application:

beforeHookedMethod Hook a method before it actually gets invoked and allows to take a look into the parameters as well as the opportunity to manipulate them.

afterHookedMethod Hook a method after it got invoked and returned. If the method is not void it is also possible to look or manipulate the return value before the actual caller method receives it.

DAMNs Xposed Module. Although Xposed was designed to make changes in behavior or look and feel of an application, it also can be used for investigations. DAMN make use of the hooking mechanism of Xposed and hook every constructor as well as all methods in any class. This makes it possible to take a look at the passed parameters, respectively the returned values. Moreover, we make use of the manipulation feature to perform changes on the running application.

[5]https://github.com/civetweb/civetweb

[6]https://github.com/rovo89

DAMNs Xposed module will be loaded with any new application. If the module finds a *packages_damn* (Section 4.5) file, it checks if there are packages declared and hooks every constructors and methods in this package. DAMN uses Java reflection [14] to find all information about a package. The *XDAMNHook* class is a customized hook which can send information to the DAMN server. It also allows to pause, single step, run or manipulate the application if the *XListener* receives data from the web interface.

4.5 Application Configuration File

Because it is possible to select single packages of an application we need a mechanism to configure our Xposed module only interacting with those. Android uses the discretionary access control (DAC [16]) to restrict the file access of applications [2]. Because the Xposed module is running as part of the investigated application it also restricted to its permissions. Since we can not assume that every application have permissions to read any file, we have to put the configuration file in the installation directory. All files in this directory can be accessed from any application without additional permissions. Our Xposed module will check if the configuration file *damn_packages* exists on startup and if there are any packages declared in it.

4.6 DAMN Runtime States

DAMN can manipulate the application in a way that it can be paused, single stepped through the interaction points or ran as usual. All those changes can be done at runtime. To single step through the application at the beginning, the first state after an application gets launched is the pause state. This makes it possible to analyze the very first actions an application performs. To perform the runtime manipulation we need interaction points where we can interact with the application. Xposed provides two different options to interact with an application at runtime. The first interaction point is raised if a method gets called. If this happens, Xposed will redirect the call to our Xposed module. The very same applies if a method returns. On both interaction points, we can pause the application, investigate the values of parameters respectively the return value and have the opportunity to manipulate them.

4.7 Runtime Manipulation

DAMN supports manipulation of values during execution. It is possible to manipulate values of parameters or return values at any interaction point. Those values are displayed as JSON objects at the interaction page and can be overridden. The reason for using JSON is the compact and simple way of displaying the information. Basically, all Java primitive data types are supported as well as byte arrays. If an unsupported type gets passed, its class name will be displayed. If the manipulated value has not the same data type as the original, DAMN will raise an exception.

5. EXEMPLARY INVESTIGATION

In order to evaluate the DAMN framework for real world applications, we investigated two quiz applications. For comparison reasons we picked one with advanced obfuscation and another without. Both applications will not be named with their actually name because it is not our intention to expose anyone in this paper.

Quiz Layout. The investigated quiz applications have basically the same way of displaying the questions and possible answers. The question will be displayed on the top of the activity as well as a counter which indicates the time left for answering. There are four buttons placed below that are holding possible answers.

Quiz A. After setting up DAMN for the investigation we start Quiz A and navigate through the app until we can start a new game. Right before we start the game, we switch into single step mode. Now we can see that the browser displays which method got called and which parameters are passed. Towards some initial calls we can observe that the questions are stored in a *sqlite*[7] database. After a short investigation over the adb shell we can see that the database entries are encrypted. But we do not need to search for the key which is either stored or hard coded in the source because we can directly observe the values from the application right before they gets displayed. After some more single stepping through quiz A we find an object which holds one string with the question and four possible answers which are unencrypted. The real interesting part happens if we look at the method that gets called if we choose an answer. The code checks if the chosen answer holds the same string as the first answer in the object we got previously. This means that we can either change the passed parameter to the method in a way it is always equal to the first entry or manipulate the return value to *true*. Both will lead to a correct answer without pressing the correct one. This manipulation was done in a few minutes on an application with more than 200.000 classes and much more obfuscated methods.

Quiz B. The second application uses less obfuscation then quiz A. We switch again to single stepping and start a new game. This time we can see that the *GCM* (Google Cloud Messenger)[8] is involved as well as *GSON*[9] to handle JSON objects. The main difference between quiz A and B is that quiz B is requesting each question directly from a server instead of storing them locally. We can detect the question as well as four possible answers but no hint which one is the correct one. The index of the chosen answer and the time it lasts to choose it is wrapped and send again to the server. After that, the server replies if the answer was correct or not. Manipulation of the time also fails as the server checks if the requested question do not pass a certain time limit. After further manipulating we came to the conclusion that it is not possible to manipulate quiz B.

Recap. DAMN made it possible to investigate two obfuscated applications in a very short period. The manipulation features DAMN is offering makes it possible to check quickly runtime behavior on different parameter values. This makes it easy to test various scenarios. Although quiz A was using very strong obfuscation and encryption, it was no problem to manipulate this application. Quiz B did not use this strong obfuscation but use another design pattern which protected it from our attempted attacks.

Despite this was only a short investigation, the outcome confirms our assumption that obfuscation do not protect ap-

[7]https://www.sqlite.org/
[8]https://developers.google.com/cloud-messaging/gcm
[9]https://github.com/google/gson

plications from being attacked. Of course, it is an additional protection which makes it harder to read an application, but dynamic manipulation tools like DAMN are making this protection obsolete. An appropriate security design is relentless for any application.

6. FUTURE WORK

During all investigations DAMN was facing multiple problems that could be solved with additional features. We are describing some of those features below.

Behavior Rules. One of the first additional feature will be behavior rules which helps to automate some tasks during the investigation. For example, those rules could be applied on specific methods and if a certain case arrives they can perform manipulation of parameters. This will bring further enhancements on testing an application under different conditions.

Multi Threading. Another useful feature would be the support of multi threading. DAMN is already supporting this feature but do not have a proper implementation to display multiple threads in the browser interface.

DAMN is freely available for non-commercial usage under the GPLv2 license. It is open source and the project can be found at github[10]. Changes in code that will extend or bring further stability are desirable.

7. CONCLUSION

During the investigation showed that DAMN could find the relevant code sections quickly. Whereas obfuscation drastically increased the amount of methods it could not prevent the performed manipulation attack. Furthermore, additional protection techniques like the encryption of locally stored data did not take affect because we can directly access the data from the application. The manipulation feature of DAMN helped to check quickly some possible attack scenarios. DAMN could confirm that security through obscurity is not guaranteed and should not be practiced. Our tool showed that obfuscation did not add any additional security protection to an application. The very same can be derived from native code parts that are used by some applications to protect information. While DAMN can not handle native libraries at the moment, other tools can.

8. REFERENCES

[1] F. Bellard. QEMU open source processor emulator. *URL: http://www.qemu.org*, 2007.

[2] D. Chell, T. Erasmus, J. Lindsay, S. Colley, and O. Whitehouse. *The Mobile Application Hacker's Handbook*. Wiley, 2015.

[3] J. J. Drake, Z. Lanier, C. Mulliner, P. O. Fora, S. A. Ridley, and G. Wicherski. *Android Hacker's Handbook*. Wiley Publishing, 1st edition, 2014.

[4] T. Eder, M. Rodler, D. Vymazal, and M. Zeilinger. Ananas-a framework for analyzing android applications. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, page 711–719. IEEE, 2013.

[5] N. Elenkov. *Android Security Internals: An In-Depth Guide to Android's Security Architecture*. No Starch Press, 2014.

[10]https://github.com/baer-devl/DAMN

[6] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[7] A. Hoog. *Android Forensics: Investigation, Analysis and Mobile Security for Google Android*. Android Forensics: Investigation, Analysis, and Mobile Security for Google Android. Elsevier Science, 2011.

[8] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, page 639–652. ACM, 2011.

[9] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. ANDRUBIS-1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[10] J. Oberheide and C. Miller. Smartphone OS Market Share, 2015 Q2 @ONLINE, 2012.

[11] P. Padala. Playing with ptrace, Part I. *Linux Journal*, 2002(103):5, 2002.

[12] N. J. Percoco and S. Schulte. Adventures in bouncerland. *Black Hat USA*, 2012.

[13] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, page 1808–1815. ACM, 2013.

[14] C. Ullenboom. *Java ist auch eine Insel: Programmieren mit der Java Standard Edition Version 6;[das umfassende Handbuch; aktuell zu Java 6; DVD-ROM inkl. Openbook-Bibliothek (4000 Seiten), 300 Aufgaben und Lösungen, Java 6 und Eclipse 3.2, viele Zusatztools]*. Galileo Press, 2006.

[15] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis: Android malware under the magnifying glass. *Vienna University of Technology, Tech. Rep. TRISECLAB-0414-001*, 2014.

[16] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *null*, page 213. IEEE, 2003.

[17] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *USENIX Security Symposium*, page 539–552, 2012.

[18] L.-K. Yan and H. Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *USENIX security symposium*, page 569–584, 2012.

[19] M. Zheng, M. Sun, and J. Lui. DroidTrace: A ptrace based Android dynamic analysis system with forward execution capability. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2014 International*, page 128–133. IEEE, 2014.