Analyzing the Flow of Sensitive Data and Facebook Permissions in Android Applications

Skyler Ellenburg

Advisor: Kristopher Micinski

Department of Computer Science, Haverford College

April 30, 2018
## Contents

1 Introduction ......................................................... 3
   1.1 Background .................................................. 4
   1.2 Motivation .................................................. 5

2 Literature Review .................................................. 6
   2.1 Google Store Scraping ....................................... 6
   2.2 Social Media Permissions ................................... 8
   2.3 Android Permission Analysis Tools ......................... 12

3 Methodology ........................................................ 18
   3.1 Sample Application .......................................... 18
   3.2 Collect Log .................................................. 19
   3.3 Detection Script ............................................. 20

4 Results ........................................................................ 22

5 Conclusion and Future Work .......................................... 25
   5.1 Acknowledgements ............................................. 26
1 Introduction

When Android applications request permission to use certain data, the extent to which they use that data and where it goes is often vague, deceptive, or completely ignored by the user. Researchers in this field examine the use of permissions in Android applications to analyze what data is being used when, where it comes from, and where it is going. Our primary goal is to contribute to understanding how we can improve regulation on permissions to request data in the most efficient and secure way, leaving the user less vulnerable and more informed. Particularly in the context of social media applications, platforms like Facebook can grant access to data that the user does not permit through Android settings, therefore creating a loophole for the flow of sensitive user data.

Applications and devices prompt the user on permission calls, but sometimes these prompts are not specific, clear, or comprehensive; these permission calls are classified as “broad.” When a permission is “broad”, the user may be granting an application more access than they realize, and sometimes this access involves sensitive data. There are existing tools that analyze the data flow in the app in order to detect the use of permissions or access to APIs and servers. Each tool contributes to a different aspect of Android security research, but few focus on social media permissions, as the use of social media has grown exponentially in the recent years. In order to analyze this data flow in the context of social media permissions, I combine the existing tools Redexer [JMF12a] and Symdroid [JMF12b] with my own analysis scripts to detect access to the Facebook API from the application code. This process and analysis described in this paper contributes to the broader research on how we can detect the use of Facebook API calls in bypassing Android permission settings.

To complete this analysis, I must understand the specific calls to Facebook’s Graph API, or application program interface, in order to search them in the application and compile a list of the fields and permissions to which Facebook grants access. In this paper, I explain the methods to create
a large data set of applications downloaded from the Google Play Store, but I ultimately built my own application to test the script. I compiled a list of other applications that access the Graph API in order to validate my permission analysis. The ultimate purpose is to contribute to an effective tool that takes an Android application as input and checks which social media permissions it uses and for what purpose, returning an analysis of the flow of sensitive data through the application.

1.1 Background

Currently, the tool Redexer [JMF12a] provides an efficient and effective process for researchers in this field of Android data flow to analyze applications, but it decompiles applications into Dalvik bytecode, which is stack-based and needs manipulation from the user to apply the code to an existing application. The implementation through Symdroid [JMF12b] uses Redexer to rewrite the application code and analyzes the log of this rewritten application to detect calls to Android permissions. My efforts lay the groundwork to expand Symdroid to detect Facebook permissions.

An existing tool Soot [ARB17] provides program analysis using java bytecode, and we will use their methods to build our tool. It is a framework used by many researchers to build credible tools, so the tasks that Soot accomplishes, such as understanding and mutating the infrastructure and analyzing data flow, will be useful to us in our detection of the flow of data from the application to the Android and social media APIs and back to the application. FlowDroid is another tool mentioned by S. Arzt et al. in their Soot paper, which tracks the application quality similar to the tool Phosphor [BK14], described by J. Bel and G. Kaiser to analyze the data flow within the application. Knowledge of Phosphor supplements our understanding of the relationship between Phosphor itself, the languages Java and Scala, and the Dalvik Virtual Machine, which interact similarly to Soot and the java bytecode.
1.2 Motivation

The increasing use of applications, specifically social media applications, is presenting new avenues for the flow of data sometimes out of our control, or at least out of sight, and users are becoming more conscious of the deception used by application developers and companies to gain access to data and personalize their applications. Android applications and devices specifically, compared to Apple Inc.’s iOS system, reach wider audiences due to their lower cost, but with this comes greater risks: unlike iOS, Android is practically open source, so anyone has access to the makeup of the Android system, inflating the potential for breaches in security.

My specific contribution to this project through my focus on Facebook permissions, which are growing in popularity of use from the application development side and decreasing in security and regulation from the user side, serves to highlight trends to bring awareness to users and summarize commonalities for researchers. Facebook specifically has recently entered the news because of their immense data set and its political implications, for it presents a risk for our nation’s security, and this publicity culminated in Facebook CEO Mark Zuckerberg’s trial before Congress in April of 2018. Because of these concerns, it is important for us to understand the flow of our data, especially within Facebook permissions. The analysis technique described in this paper is targeted towards other researchers who want to find and analyze trends in the use and misuse of permissions in order to prevent and better-regulate breaches in application security. They can apply this tool to other categories of applications, specific types of data, and any other significant fields that they hope to analyze.
2 Literature Review

In order analyze these applications, we must understand how existing tools find the use of Android or social media permissions within applications, gain access to a large data set of applications for testing, and obtain results through certain algorithms or analysis. Therefore, the relevant work for this project includes descriptions of existing techniques to scrape the Google Play Store or other application markets for popular Android applications to create large data sets. We also gain a deeper understanding of social media permissions, specifically within Facebook’s Graph API, and how tools detect and examine how applications can gain information on the user by bypassing the intended Android permissions and using social media permissions instead. Lastly, the relevant work includes papers on Android permissions specifically and the tools that find and analyze the use of these permissions.

2.1 Google Store Scraping

Two existing tools that utilize scraping to analyze the success of their implementation are PScout [AZHL12] and Edgeminer [CFB +15], and in their project that examines calls to the Facebook API specifically, M. Frank et al. train their model with Android and Facebook application data from the Android market. It is important to test on a large data set of applications, but due to Google’s terms regarding unconventional downloads from the Play Store, I did not attempt scraping. Although I did not implement these techniques myself, understanding how to compile a large data set would aid further testing of this project and others, and Google does encourage certain prohibited processes if their purpose is research rather than commerce.

In [FDFS12] to collect Android and Facebook application data for training their model, M. Frank et al. scraped the web version of the Android Market for 188,389 Android applications, which was 59% of the Android market at the time, in 2011. They first crawled the lists of “top free” and
“top paid” applications because they were looking for applications of high-reputation, which they define as applications with at least 100 user ratings of 4 or higher. They then found links to 156,283 other applications through search by 1,000 randomly-selected dictionary words and all possible two-letter permutations. These searches provided links to the description pages, which they then parsed for application names, categories, ratings, and permissions. For the Facebook data, they found a set of 27,029 applications through the same strategy that Chia et al. used in [CYA12]: crawling SocialBakers site [Soc], which links to Facebook applications from which they collect statistics and analytics. This large dataset of Android and Facebook applications enabled M. Frank et al. to run their algorithm and train their model on a data set comprehensive enough that its results can apply to the Android Market as a whole.

In terms of existing tools that also scrape application resources, we can examine the work done for PScout [AZHL12] and Edgeminer [CFB+15] to build on their strategies and download applications for testing our own tool. To create their tool Edgeminer, Cao et al. analyze and scrape versions 2.3, 3.0, and 4.0 of the Android Framework to extract all of the flow transitions from the codebase and return a list of registration-callback pairs to contribute their security analysis through a control flow graph. They first scrape a list of potential callbacks and identify all callsites in the framework, which we can draw parallel to having a list of potential application names or ranks and then places to look in the code for those names or rankings. They then run a backward data flow analysis for each callsite to find the location of the site, and if the callback is accessed at the suggested callsite, this completes a registration-callback pair returned by the analysis, thereby detecting an aspect of the overall flow of the framework. Finding pairs of application names or rankings and links to download them or access their code will be useful in our own analysis, so the methods from [CFB+15] can be applied to our analysis of the Google Play Store. In their creation of the tool PScout, described in [AZHL12], Yee Au et al. also analyzes Android versions, spanning from 2.2 to
4.0, the most recently released at the time of their research in 2012. They mention the size of the Android framework as a challenging aspect for the extraction of permission data, which is similar to the daunting size of the Google Play Store, and they scale it down by first identifying and labeling permission checks in the framework, then building a call graph like the researchers do for Edgemin, and then performing a backwards reachability traversal on the graph in order to detect pairs of API calls and the permission checks that they potentially reach, repeating this analysis until the number of permission checks converges. Therefore, according to the execution descriptions of these tools, we should scrape the Google Play store through a backwards traversal of a graph or a backward data flow analysis to find pairs of detectable application criteria with links to downloadable versions of the applications to then scrape the store.

2.2 Social Media Permissions

Another component of the project is analyzing calls to the Facebook API to determine which are most common and dangerous. From the Graph API documentation [Devc], a call to the Facebook API looks like GET graph.facebook.com, but once we find that this line of code exists in application code, I want to expand the search of my script to which permission calls are being made and when. Specifically I am focused on which Android permissions are usually bypassed through the use of the social media permissions, granting apps access to information in an indirect way that the user does not expect. The papers that provide insight towards this portion of my research give clarity on common permission request patterns, with a focus on Facebook applications and permissions, and an analysis of user opinions, which guide our approach in detecting calls to the Facebook API specifically and the experience of the user.

One source of information on finding patterns in permission requests, specifically with examples from applications that access Facebook permissions, is [FDFS12], through which M. Frank et al.
find overlapping clusters of permissions from a series of applications to compare the Facebook and Android permission requests within applications of varying quality, measured by user satisfaction and popularity. Using the Android and Facebook application data that they scraped from the Android Market from Section 2.1, they investigated properties of the applications, from which the permissions used is the most relevant to this project. Table 1 below shows the 15 most frequently requested Facebook permissions along with their frequency and a description from the Graph API documentation [fd], with some of them now disabled since their paper was published in 2012.

Once they compiled the data set and found the most frequently requested permissions, M. Frank et al. use matrix factorization and clustering to mine the data for statistically significant permission request patterns. Their approach uses unsupervised learning because the alternative strategy requires manually analyzing and labeling applications based on their properties, which is time-consuming and may not be as accurate, as labeling can be restrictive for a large, diverse set of applications. They map $N$ applications to $D$ permission requests in a binary matrix $x \in \{0, 1\}^{N \times D}$, where $x_{id} = 1$ signifies that application $i$ requests the permission $d$. Each row of the binary matrix represents the permission requests for some application $i$. Their technique, which they call pattern mining, takes this matrix $x$ as input in order to find the matrix $u$ that encodes which permissions are commonly requested together, which is called the permission request pattern, and the matrix $z$ that encodes which applications have the same request patterns. In their model, for $K$ as the number of patterns found, then $z \in \{0, 1\}^{N \times K}$ and $u \in \{0, 1\}^{K \times D}$. The goal is to find the boolean product of $z$ and $u$ which is

$$c = u \otimes z \text{ such that } c_{id} = \bigvee_{k=1}^{K} (u_{ik} \land b_{kd})$$

This product will give the factorization, so they define their goal as finding a factorization $(z^*, u^*)$ which approximates $x$, so $x = u^* \otimes z^*$.

They then use this binary matrix factorization to create a model that computes the likelihood
<table>
<thead>
<tr>
<th>Frequency</th>
<th>Permission Name</th>
<th>Description of Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>67.35%</td>
<td>basic</td>
<td>Public profile, friends, and email</td>
</tr>
<tr>
<td>23.12%</td>
<td>publish_stream</td>
<td>Replaced by publish_actions</td>
</tr>
<tr>
<td>13.93%</td>
<td>email</td>
<td>The user’s primary email address</td>
</tr>
<tr>
<td>3.38%</td>
<td>user_birthday</td>
<td>The user’s birthday</td>
</tr>
<tr>
<td>2.47%</td>
<td>offline_access</td>
<td>Facebook has since removed this permission</td>
</tr>
<tr>
<td>2.12%</td>
<td>user_photos</td>
<td>The photos a user has uploaded or been tagged in</td>
</tr>
<tr>
<td>1.79%</td>
<td>publish_actions</td>
<td>Publish posts and other activity on behalf of the user</td>
</tr>
<tr>
<td>1.52%</td>
<td>user_location</td>
<td>The user’s current city</td>
</tr>
<tr>
<td>1.32%</td>
<td>read_stream</td>
<td>Facebook has since removed this permission</td>
</tr>
<tr>
<td>1.16%</td>
<td>user_likes</td>
<td>The list of all objects that a user has liked</td>
</tr>
<tr>
<td>1.05%</td>
<td>user_about_me</td>
<td>The user’s personal description from ‘About Me’</td>
</tr>
<tr>
<td>0.95%</td>
<td>friends_photos</td>
<td>Facebook has since removed this permission</td>
</tr>
<tr>
<td>0.75%</td>
<td>user_hometown</td>
<td>The user’s hometown location</td>
</tr>
<tr>
<td>0.69%</td>
<td>user_videos</td>
<td>The videos a user has uploaded or been tagged in</td>
</tr>
<tr>
<td>0.6%</td>
<td>friends_birthday</td>
<td>Facebook has since removed this permission</td>
</tr>
</tbody>
</table>

Table 1: The fifteen most frequently requested Facebook permissions out of 62 from [FDFS12] and their descriptions from the Graph API documentation [fd]

that a given factorization \((z,u)\) represents the statistically significant patterns of \(x\) by tweaking the entries of \((z,u)\) to produce an outcome \((z^*,u^*)\) that is not an exact representation of \(x\) but a close approximation. The matrix \(u^*\) shows the assignment of permissions to request patterns, where one permission can appear in multiple patterns, while the matrix \(z^*\) assigns these patterns to applications and clusters the applications into groups that request the same permission patterns.
The number $K$ of patterns is provided as an input to the algorithm, described in Section 4.1 of their paper, and depending on the complexity of a dataset a small value for $K$ can give accurate results while a larger value of $K$ might cause overfitting, and they use an instability analysis to find the best $K$ for their dataset, which was $K = 5$ for their Facebook data set. They trained the model on high-reputation applications, with 400 Facebook applications used for testing and 1,598 applications used for training.

Through their tests, M. Frank et al. found that these Facebook permissions fit into five patterns of requests, which is useful for my research for calls to the Facebook API, as these are key methods to look for. The five groups of patterns are basic; basic and publish_stream; basic and email; basic, publish_stream, and user_photos; and basic, email, user_birthday, and user_location. The request patterns signify that these permissions are commonly used together, so not only can we use their analysis strategies to find permissions using our own tool, but we can use these patterns to understand how these applications likely access the Facebook API, especially if we are looking at a certain category of application, as M. Frank et al. find similarities in request patterns within a particular category of applications.

In terms of their model and results, the paper describes the false positive and false negative rates and their relationship with the characteristic of the application. A false positive result from the model is a permission request pattern assignment where the application does not actually contain all of the permission requests in the pattern, and a false negative result is a failure to recognize any of an application’s permission requests within the pattern given. For the Facebook applications tested, 2% of the high-reputation applications have at least one false positive, but almost none have more than one, and under 20% have at least one false negative, with fewer than half of those applications having more than one false negative. Their data for the high-reputation test set aligns with the training set, so the results found are reliable and can reflect what we should expect from
all high-reputation Android applications. Applications of lower popularity have different request patterns from those of the high-reputation applications, and they have higher rates of false positives and false negatives, which affirms that the high-reputation applications are actually classified as higher-quality applications by the model. The paper uses its results to emphasize that permission request patterns can help categorize the application and even determine the quality or category of the application. Another useful aspect of the research of M. Frank et al. is its reference to [EOM09], in which Enck et al. created a tool that detects a blacklisted set of permissions, which are useful in our search for permissions to detect with our tool, as well.

To analyze application behavior through application profiles along with studies of the user experience, S. Rosen et al. in [RQM13] use a knowledge base that they created and map API calls and “privacy-related” behaviors, which we would call permissions. Using static analysis, these profiles indicate what the application is capable of and what it actually shows the user, which are often very different results that leave users uneasy, realizing that an application is deceptive. When they showed the users these profiles, the users gained more understanding of the security, or lack thereof, of the applications; in a case study on the Facebook application, relevant to our studies of the Facebook API, S. Rosen et al. found that its behavior specifically with user location, video, and phone number left users uneasy because of the breaches in privacy that those permissions cause, though these do only run while the application is active rather than while it is in the background. This paper serves to show that a broad analysis of the privacy behaviors, rather than just a focus on the permissions, along with the user opinions can point to the breaches in security within applications.

2.3 Android Permission Analysis Tools

Researchers have developed tools to detect Android permissions and potential or actual misuse of these permissions within application calls to the Android API or other databases. Papers on the
topic of application privilege, or the amount of power the user gives to an application through accepting broad permissions, can provide us with examples of specific Android permissions that leave the user vulnerable, which we can apply to our own research on which permissions to look for in our detection tool. We can also gain more detailed descriptions of the structure of the Android API to understand the calls to the API, the permissions most commonly used, and the potential replacements to enhance the security of the application.

In [FDFS12], through the same processes for Facebook applications, M. Frank et al. find the 15 most frequently used Android permissions to run their model and mining algorithm, detailed in Section 2.2. These permissions are outlined below in Table 2, and they help us understand not only which permissions will likely be in the application code that we run through our tool but also show which permissions users may bypass through the use of social media permissions. The results of the Android application analysis by M. Frank et al. include probable permission request patterns returned by from the algorithm and rates of success of their model, which has higher error rates for the Android applications tested than the Facebook applications. According to M. Frank et al., these error rates indicate the complexity of the Android dataset. For example, out of the 173 total Android permissions, many are “infrequently-requested permissions”, so they are not a part of the permission request patterns within the study, which increases the false negative rate of the model. From this paper we can generalize the complexity of the Android data set, as apps of a certain category or quality have similar Android request patterns, though the frequency depicted in Table 2 is the most relevant in terms of our tool.

The Android permissions documentation [Deva] states that permissions are classified as “dangerous” if they alter the settings of the device or breach the users privacy. The documentation lists these dangerous permissions, which we can compare to the frequently used permissions and those found in the application code that we test. Frequently used and dangerous permissions from Table 2
<table>
<thead>
<tr>
<th>Frequency</th>
<th>Permission Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>69.76%</td>
<td>Network communication: full Internet access</td>
</tr>
<tr>
<td>43.24%</td>
<td>Network communication: view network state</td>
</tr>
<tr>
<td>30.26%</td>
<td>Storage: modify/delete USB storage</td>
</tr>
<tr>
<td>26.47%</td>
<td>Phone calls: read phone state and identity</td>
</tr>
<tr>
<td>18.34%</td>
<td>Your location: fine (GPS) location</td>
</tr>
<tr>
<td>16.89%</td>
<td>Your location: coarse (network-based) location</td>
</tr>
<tr>
<td>16.16%</td>
<td>Hardware controls: control vibrator</td>
</tr>
<tr>
<td>15.01%</td>
<td>System tools: prevent device from sleeping</td>
</tr>
<tr>
<td>8.22%</td>
<td>Network communication: view Wi-Fi state</td>
</tr>
<tr>
<td>8.11%</td>
<td>System tools: automatically start at boot</td>
</tr>
<tr>
<td>6.71%</td>
<td>Services that cost money: direct call phone numbers</td>
</tr>
<tr>
<td>6.27%</td>
<td>Your personal information: read contact data</td>
</tr>
<tr>
<td>5.59%</td>
<td>Hardware controls: take pictures and videos</td>
</tr>
<tr>
<td>4.61%</td>
<td>System tools: set wallpaper</td>
</tr>
<tr>
<td>3.9%</td>
<td>System tools: retrieve running applications</td>
</tr>
</tbody>
</table>

Table 2: The fifteen most frequently requested Android permissions and their actions from [FDFS12] include **READ_PHONE_STATE**, **ACCESS_FINE_LOCATION**, **ACCESS_COARSE_LOCATION**, and **READ_CONTACTS**. Through the documentation we also find guidelines for how to alert a user of a dangerous permission request within the application, but sometimes if a dangerous permission has already been granted by the user, the application is not required to prompt the user again for another dangerous permission in the same permission group; for example, if the application is granted the **READ_CONTACTS** permission, shown in our chart to have a frequency of 6.27%, if the application follows with a re-
quest to WRITE_CONTACTS, not listed in Table 2, the Android system immediately grants that same permission. This provokes the question of whether or not the user should be prompted for such a request and if this should be seen as efficient rather than deceptive.

Transitioning to specific tools that detect dangerous Android permissions in applications, in [JMV+12] Jeon et al. describe their strategy to analyze the “deviation from least privilege” that threatens the secure flow of data through applications that gain more access than they need by means of broad or vague permissions, in that they do not explain clearly to the user what data is being used and when, and their solution comes in the combination of two tools, Dr. Android and Mr. Hide. The two tools combine to replace the permissions within the app with more specific and accurate permissions, where Mr. Hide is a service that enables access to the sensitive data for Dr. Android to remove and replace the permissions with those created by Mr. Hide based on the use and flow of this sensitive data. They name the following advantages of their approach over other tools before them that serve a similar purpose: jailbreaking is not necessary and their strategy accounts for the constant evolution of apps through its adaptable and fluid permissions as opposed to fixed permissions. They identified seven of the most dangerously broad permissions subject to breaches in security and used their tools to find fine-grained replacements, which resulted in stronger privacy for their tested applications. These seven “dangerous” permissions include INTERNET, READ_PHONE_STATE, WAKE_LOCK, ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION, WRITE_SETTINGS, and READ_CONTACTS, which line up with the frequently used dangerous permissions in Table 2, and the paper lists replacements for each, which are ultimately the results of the tools Mr. Hide and Dr. Android. These replacements do not modify the functionality or performance of the applications, but only improve security through limiting the flow of sensitive information enabled by broad permissions.

Similarly, while elaborating more on the technical side of Android Permissions and the struc-
ture of data flow through Android applications, Porter Felt et al. share another tool in [FCH+11] that detects these API calls linked to dangerously broad permissions, what they call instances of “overprivilege,” and maps them to new permissions. Their tool called Stowaway has detected overprivilege in one third of the Android applications they tested, and they found that often these lapses in security are due to poor API documentation. The strategy that Stowaway uses differs from the strategy in [JMV+12] because it detects the API calls and finds the permissions within the permission map that would be appropriate for that call instead of detecting the permissions and removing and replacing those. This paper also focuses more on providing background for the Android API structure, going in-depth about the three steps of an Android API call: first the application calls on the public API in the library, then that library calls on a private interface within the library, and last this private interface requests the desired operation of the API call from the system service. In terms of Android permissions, the paper describes the lack of permission checks built into Android, which leaves the user vulnerable to more access to their data than they would expect; Porter Felt et al. explain that there is not a reliable system that checks the presence of specific permissions, so placing these permission validation calls through Stowaway is essential for ensuring the use of specific permissions and reducing overprivilege.

Lastly, the most important tool to this paper is Symdroid [JMF12b] created by Jeon et al. that receives the rewritten application file (.apk) from Redexer and builds it on the device, collecting a log of all activity and outputting the log in a readable format for Hogarth, its log analysis tool, to interpret the use of Android permissions in the application log. It goes beyond Redexer [JMF12a] to achieve a more static and dynamic analysis of Android applications, with a focus on their calls to Android permissions and the flow of data through the application, specifically through action methods like `.onCreate` and `.onClick`. Through these methods, Symdroid outputs a JSON object mapping the basic blocks of activity code from the log to the Android permission uses, providing an
accurate mapping of data flow.
3 Methodology

To tackle this problem of overprivilege in Android applications through the use of deceptive Facebook permissions, I created an application that uses the common Facebook permissions from my research in Section 2.2, rewrote the app in Redexer [JMF12a], and then collected and analyzed the application log with Symdroid [JMF12b] and my own script analysis. My enhancements of the Hogarth scripts search for Facebook permission calls in the application log, which could detect the overprivilege, defined in Section 2.3, and deceptive social media permission calls that bypass the Android permission calls.

3.1 Sample Application

The sample Android application, called HogarthTest [Ell], serves as a test to my Symdroid log analysis supplements. There are existing samples on the Facebook Developer site [Devb], but I wanted to develop my own application to fully understand how Facebook permissions are accessed and used within the application code to aid in the writing of my scripts. HogarthTest accesses the public_profile permission, listed under the permission basic in Table 1. The application builds and runs on my test device, a Motorola G4 Play running Android version 6.0.1 (Marshmallow API 23).

It has a welcome screen, as you can see in the figure 1, from which you must login to Facebook to access the following app screen. Pressing the Facebook login button sends the user to the login screen in their default browser. Then they are sent to Facebook’s permission screen, which requests only the public_profile permission in a dialog box shown in the figure 1. According to the Facebook developer documentation, the public_profile permission includes the following properties of the user object: id, cover, name, first_name, last_name, age_range, link, gender, locale, picture, timezone, updated_time, and verified [Devc]. Once the user accepts the public_profile per-
mission, it sends them back to HogarthTest into the next screen of the application. The call to the permission is executed on the `LoginManager`, which keeps track of the permissions set by the user in the line below. The specific call is through the method `logInWithReadPermissions` which takes as a parameter the permissions that the app requires as a list of strings. This method and the permission strings are consistent for any permission calls at login, so their existence in the app code proves a call to Facebook permissions. To access the specific user data, the application uses a Graph API request with the specific `AccessToken` for that user, so the `GraphRequest` method also gives us insight on the flow of data from the Graph API and through the application.

### 3.2 Collect Log

Through Symdroid, the researchers collect logs of applications and run them through the tool to test its success. Once I collect the log through Redexer and Symdroid for my application HogarthTest,
I search that log for the specific methods described in Section 3.1, which can then be applied to Symdroid’s current scripts that detect Android permissions only. The application log contains a list of all actions executed manually by the user, so if the user presses the Facebook login button and accepts the `public_profile` permission, for example, that call will appear on the log after the process of collecting the log is complete.

After downloading the Symdroid repository [JMF], I first instrument the application with one of Symdroid’s Ruby scripts, which use Redexer to rewrite the application to include more specific logging instructions. Then I collect the log of the rewritten application activity with another Symdroid script while I execute every action of the application. Once the log is complete, I exit Symdroid, and the log outputs to a text file in the application directory `testuserlog` in Symdroid. The log can also be found on the devices memory in its SD card, which I access in my detection script instead of the Symdroid logging directory.

### 3.3 Detection Script

Since I did not use Symdroid’s Ruby Android permission analysis and instead wrote my own script in Python to detect Facebook calls specifically, I first must pull the logs written onto the phone by Redexer. They are stored in the `sdcard` directory within the phone, so my script first pulls all log files from the device. It then concatenates the logs into one large log of all activity, since the complete activity on the device, from the start of log collection until completion, is defined within multiple log files. Then my script parses the concatenated log, looking for instances of `loginWithReadPermissions` and `GraphRequest` to determine if the user was prompted with Facebook permissions, which permissions the app requests, if the user accepted those permissions, and where the user’s data is being accessed within the application.

Symdroid’s Ruby script generates a JSON object mapping the basic blocks of activity code
from the logs to their Android permission uses, and my script similarly returns a JSON object containing whether or not permissions were called (“called”), whether or not data from that permission’s access is used later in the application activity (“used”), and the lines of the log in which loginWithReadPermissions (“readperm”) and GraphRequest (“request”) take place. Following the pattern of GraphRequest calls returned within the JSON object can depict the flow of Facebook data through the activity on the device. If the script detects loginWithReadPermissions but no GraphRequest, then permissions are requested but no user data is used within the application. On the other hand, if the script detects GraphRequest but no loginWithReadPermissions, the application is getting access to user data through Facebook without the users permission. If the script detects both, then the application requests permission and accesses user data.
4 Results

After running and adapting my script based on its results for HogarthTest, I tested this log collection and script process on four other Android applications that also access the Facebook API to detect their permission calls and flow of user data. Although these samples from Facebook’s developers site [Devc] make up only a small data set, it covers various permission calls and aided in my understanding of running these scripts manually, a process which would be automated and scaled up on a larger data set. The list of applications are listed in Table 3 with the permissions they call, and the field “Detected” shows that my script detects `loginWithReadPermissions`, which is the permission call, or `GraphRequest`, which is the request to user data. The descriptions below show that my script functions as expected for applications that call Facebook permissions, Android permissions, or both.

<table>
<thead>
<tr>
<th>App</th>
<th>Permissions</th>
<th>Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facebook Login</td>
<td><code>public_profile, email</code></td>
<td><code>loginWithReadPermissions, GraphRequest</code></td>
</tr>
<tr>
<td>Hello Facebook</td>
<td><code>public_profile, publish_stream</code></td>
<td><code>loginWithReadPermissions, GraphRequest</code></td>
</tr>
<tr>
<td>Messenger Send</td>
<td><code>[photos/media/files]</code></td>
<td><code>GraphRequest</code></td>
</tr>
<tr>
<td>Places Graph</td>
<td><code>[coarse location], public_profile</code></td>
<td><code>loginWithReadPermissions, GraphRequest</code></td>
</tr>
</tbody>
</table>

Table 3: The four Android applications which call the Graph API that I use to test my script

For the application “Facebook Login” (FBLoginSample.apk), my script detected both `loginWithReadPermissions` and `GraphRequest` methods. Through manually searching the activity of the application, I found that it calls to the `public_profile` and `email` permissions upon login. It can gain access to all of the `public_profile` data, including: the user key, email, and permissions granted. It also has access to posting on the timeline, though it does not request the `publish_stream` permission, but this part of the application did not actually post on my timeline, so I am not sure of the functionality there,
and therefore do not deem this deceptive.

In testing my script with “Hello Facebook” (HelloFacebookSample.apk), upon login the application requests access to the public profile permission to use the user photo and name on the home screen. The user can share posts and post photos from the application, and enabling this feature requires accepting the publish_stream permission on click. My script detected both of these calls to permissions as well as the GraphRequest calls, which are used to display the profile photo and to pass on the user key to post.

The “Messenger Send” application (MessengerSendSample.apk) extends to the Facebook Messenger application, which requires user login without permissions. The app enables sending a photo in a message through Messenger, and since there are no permission prompts, the script should only detect GraphRequest methods because of the activity after leaving the application. The only permission request it requires is Android’s Photos/Media/Files permission in order to pass the photo through the device to Messenger. My script results are consistent with this because “called” is false, since no Facebook permissions are called, while “used” is true, indicating only the use of GraphRequest. This is not deceptive because the user logs in via the Messenger app, but it does support the effectiveness of my script in detecting permission use and data flow.

Upon opening the app “Places Graph” (PlacesGraphSample.apk), it requests the user’s Coarse Location, a dangerous Android permission. It does not require the Phone Android permission, but it does give the option to approve it, which writes and reads call log and phone state, deemed dangerous in Section 2.3. Upon login, the application requires the users public profile, though it is unclear why - this could be a potential breach in security because through this permission the application has access to email and user data, which it could give to Google Maps or other features within the application. My script detects both loginWithReadPermissions and GraphRequest methods because the Facebook permission is called and the data is used, though we do not know
how, since the purpose of the application is to use a Google Maps search and not necessarily any user data.

Therefore, through testing my script on these four applications, I can confirm that my script detects the permission calls and flow of data through GraphAPI requests.
5 Conclusion and Future Work

In order to understand where our data is sent and for what purpose, the Android and Facebook APIs give us prompts to indicate calls to permissions, but sometimes data from these permission calls can be passed on to other applications, databases, and features that may not be clear to the user. In cases of sensitive data transmission, this can present security breaches, which we work to detect and prevent. Through this research, professionals create tools to draw conclusions about deception and general trends amongst applications, and this project serves to contribute to that research, specifically in the context of social media application through a focus on Facebook.

Future work or enhancements of this project include implementing the strategies of my script in Symdroid’s code. My script detects Facebook permission calls, so enhancing Symdroid’s current Android detection with my Facebook detection strategies will create a social media analysis component to Symdroid’s data flow analysis. Along with that, expanding this research to the Google and Twitter APIs, along with any other popular social media or Big Data powerhouse, can create a comprehensive social media analysis. In terms of fine-tuning the script analysis, an enhancement would be editing Redexer logging to include strings, which would show the exact permissions called in the log - my script detects the existence of a permission call, and my analysis determines if that is consistent with the prompts given during my manual activity log, but Redexer logging scripts can be modified to show the specific permission strings, proving that certain permissions are called but not others. One last addition would be searching for moments in the log where the application is in the background, but the data from that application is still being transmitted through Graph requests. A current security issue is applications accessing user data while in the background, so this could be examined through the logging process.
5.1 Acknowledgements

Thank you to my advisor Kristopher Micinski for his guidance and assistance. I also appreciate the work of the professors in Haverford’s Computer Science department in providing resources and seemingly limitless energy and effort towards me and my peers. Thank you as well to all of the other Computer Science majors and students who have made my experience as a student and a Teaching Assistant fulfilling and worthwhile.
References


This paper serves to describe Soot, one of the code bases we use to develop our tool because of its use of java bytecode and the methods it contains that can scan through Android application code.


In order to test this tool PScout, they extracted applications from the Google Android Market, so this paper provides us a strategy with which to do so for our own research, as we must download thousands of applications to test our tool.


This paper describes Phosphor, a tracking system for the Java Virtual Machine that analyzes the data flow within the application. This increases my understanding of the relationship between this tool and the languages Java and Scala and the Dalvik Virtual Machine, which we will be interacting with.

In order to test this tool Edgeminer, Y. Cao et al. downloaded applications from the Google Play marketplace, so this paper provides us a strategy with which to do so for our own research, as we must download thousands of applications to test our tool.


This paper provides a study on application permissions and the detection of potential privacy risks of applications. They analyze applications on popularity, ratings, and permissions and then search for the effectiveness of current risk detection, explaining common tricks and deception within applications.


This is the documentation for requesting Android permissions, which we use to research the dangerous permissions defined by Android and understand the function of each permission.

This site contains the documentation for building an Android application that accesses the Facebook Graph API through the use of permissions and other features.

[Devc] Facebook Developers. Graph api.

This site contains the documentation for Facebook code, including accessing permissions and other user information.


This Github repository contains my sample application for this project, which accesses Facebook permissions and tests the Symdroid logging for detecting the call to Facebook permissions.


In this paper, Enck et al. describe their tool that detects a blacklisted set of permissions, which are useful in our search for permissions to detect with our own tool.


This paper serves to describe a tool that detects the causes of “overprivilege” in compiled Android apps to prevent these applications from capitalizing on trusting users who leave themselves vulnerable to more access to their data than
they would expect. I will be using it mostly for an in depth understanding of Android permissions and the structure of Android data flow.


This is the documentation for Facebook’s Graph API, which we use to re-
search the objects defined by Facebook in its code and the descriptions of the permissions that Facebook offers application developers.


This paper is source of information on finding patterns in permission requests, specifically with examples from applications that access Facebook permissions.


This site contains the documentation for building an Android application that accesses the Facebook Graph API through the use of permissions and other features.


We are modeling our tool after Redexer, which uses Dalvik bytecode, while we plan to use java bytecode to achieve the same analysis on applications.

I used Symdroid to collect and analyze logs of applications, including my test Facebook application, which show the activity in the app and reveal the use of certain actions and permissions.


This paper describes two tools that work to fix the “deviation from least privilege” of Android applications where they gain more access to data than they require. I will use this paper to understand Android permissions and rewriting bytecode.


For my research in calls to social media APIs, this paper shows an analysis of application behavior through application profiles along with studies of the user experience. S. Rosen et al. use a knowledge base that they created and through static analysis create profiles that indicate what the application is capable of
and what it actually shows the user, which are often very different results that leave users uneasy, realizing that an application is deceptive.


The Socialbakers cite has a database of thousands of Facebook applications which have been scraped by certain tools and used for testing.